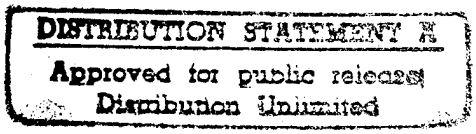
N00014–93–C–0213
FIFTH QUARTERLY REPORT

FORMAL SYSTEMS DESIGN & DEVELOPMENT, INC.
P.O. BOX 3004
AUBURN, AL 36831–3004
(205) 887 9444



N00014–93–C–0213
FIFTH QUARTERLY REPORT

Table of Contents:

FormalSystems

N00014–93–C–0213
Fifth Quarterly Progress Report

Michael Goldsmith
Formal Systems Design & Development, Inc.

April 25, 1995

Summary

This Document summarizes the progress to date in the Office of Naval Research SBIR Project N00014–93–C–0213 *Embedded Transputer-based System Design* and indicates the expected direction of the Research and Development in the following periods.

1 Overview

The level of effort expended was broadly on track during this period both at Formal Systems and at the Charles Stark Draper Laboratory (Draper) and Formal Systems (Europe) Ltd. Most of the slippage against plan reported last quarter had indeed been made up by the end of January, although actual delivery of reports was not achieved at that time. There have again been staff resource difficulties in the first quarter of 1995, with the result that some slippage of Q6 and Q7 deliverables is anticipated. The original schedule had considerable slack in the activities planned for Q8, to allow for just such an eventuality, so this should not impact the scope of the work completed within the project.

The main areas of activity and achievement during this period were:

- Continued experimentation with **FDR 2**, in parallel with its continued development by Formal Systems (Europe) Ltd, leading to appraisal and feedback into the design.
- Distillation of this experience into requirements for translation and interface tools, both for increasing the facility of expression of real-time specifications, and for increasing the range of implementation notations from which behaviors amenable to analysis can be (semi-)automatically abstracted.
- Discussion with Draper and consequent revision of the models of scheduling and architecture of the Transputer Fault-Tolerant Processor node.
- Acquisition of timing requirements for the demonstrator scheduler, based on potential application systems.
- Preliminary investigations of routes to incorporate a limited degree of continuous real-time analysis into the **FDR 2** framework.

These are detailed in the following sections and the accompanying Deliverables.
The current status of Deliverables is summarized in Table 1.

Deliverable		Due	Status
D2.1	Detailed natural-language problem statement	End Q1	Delivered Q2
D2.2	Formalization of single-lane scheduling problem and of fault tolerance requirements	End Q1	Delivered Q3[1]
D1.1	Initial requirements definition for real-time modeling extensions to FDR	End Q2	Delivered Q2
D2.3	Idealized (single-lane) scheduler model	End Q2	Delivered Q3[1]
D2.4	Fault models and redundant scheduler correctness criteria	End Q3	Delivered Q3
D1.2	Prototype software for discrete real-time extensions to FDR	End Q4	Delivered Q4
D2.5	Initial process-algebraic solution and Draper appraisal of scheduler models	End Q4	Revised Q5
D1.3	Prototype Software for Continuous Real-Time Extensions to FDR	End Q5	Deferred[2]
D1.4	Appraisal and Revised Requirements for Discrete Real-Time Extensions to FDR	End Q5	Expected Q6
D1.5	Translation and Interface Tools Requirements Definition	End Q5	Delivered Q5
D2.6	Timing Requirements Analysis for Scheduler	End Q5	Expected Q6
D1.6	Appraisal and Revised Requirements for Continuous Real-Time Extensions to FDR	End Q6	Deferred[2]
D1.7	Prototype Software for Translation and Interface Tools	End Q6	On schedule
D2.7	Initial Prototype Transputer/occam Implementation and Verification of Conformance	End Q6	Starting Q6
D1.8	Revised Code and Full Draft Documentation/Justification of Tools	End Q7	Not yet started
D2.8	Revised Prototype Transputer/occam Implementation and Architectural Specification of Potential VLSI Realizations	End Q7	Not yet started
D1.9	Final Report on Theoretical and Software Tool Developments	End Q8	Not yet started
D2.9	Final Report and Appraisal of Fault-Tolerant Scheduler Demonstrator	End Q8	Not yet started

Table 1: Deliverable schedule

Note 1: Deliverables D2.2 and D2.3 were consolidated into a single document.

Note 2: But see the discussion in §2.1 below.

2 Theory and Software Tools

The major goal of this project is to establish a viable route from specifications in Hoare's *Communicating Sequential Processes* (CSP) [5] and its real-time variants [8, 3] to implementations of real-world, substantial real-time and/or fault-tolerant systems. The initial concentration of effort under this head has been directed towards closing the gap between the current real-time specification and hand-crafted verification available within Timed CSP, on the one hand, and the available highly efficient mechanized verification and development aid for untimed CSP systems which is presented by the Formal Systems (Europe) Ltd model checking tool, **FDR**, and the new generation **FDR 2**.

The apparent tractability of the kind of problems arising from the Demonstrator Application under the discrete modeling of time is such that we have been concentrating our software development and experimentation on that approach for the present. We proposed to defer the study of continuous real-time tools until later in the project; thus far this has taken the form of a review of past approaches within the field, with an eye to their potential for adaptation to Timed CSP mechanization. An outline of the tentative conclusions to date is given in the following section.

2.1 Model-checking continuous real-time processes

There are obviously fundamental difficulties in trying directly to model and model-check continuous real-time systems. Even if the time domain is restricted to the rational numbers, rather than the reals, the cardinalities involved in a straightforward encoding of the state-space covering all possible evolutions of a system are intractably infinite.

There are two general classes of strategy that address this problem, both of which may be necessary to achieve a practical mechanization of reasoning in this field:

- Restrictions on the expressive power of the real-time language;
- Identification of temporally "equivalent" configurations of the system, and calculation modulo this equivalence.

The former technique is undoubtedly needed to some extent, since language inclusion for unrestricted timed automata is formally undecidable. The latter deprives the analysis of precise numerical data in the case of a counterexample, but does throw up the essential behavior of each component relative to the critical timings in its peers.

Based on work by Dill [4] and Lewis [7] extending state-graphs with timing constraints in a continuous model of time, Alur, Courcoubetis and Dill have studied the problem of model-checking timed ω-automata for langauge inclusion and against temporal logic formulae [2, 1]. The principal language restriction, which is unlikely to be problematic in real examples, is that the system should compare each of its finite number of clocks only with integer constants, and that the only discontinuities in their evolution should be "reset" events restarting one or more from zero. They observe that, in a finitely expressed system, there is a bound (K_c, say) on the values with which each clock c can be compared, so that the integer part of a clock's state can be modeled by a value drawn from $\{0, \ldots, K_c, K_c+1\}$,

where "incrementing" the largest value leaves it unchanged. For the fractional part of the clocks, the only effects that might be detected from the untimed observation of state transitions are those due to the relative order (or synchrony) of crossing integer boundaries; adequate timestamping can therefore be abstracted by observing which clocks have most recently "ticked". Thus the significant timing information can be encoded as the product of the (K_c+1) terms and the number of ordered partitions of the clocks; while potentially very large, this factor need not be prohibitive of mechanized exploration.

Jackson's doctoral work [6] presents a finitized dialect of Timed CSP, which exhibits the same properties. We have been giving some thought to how these notions can be implemented within the **FDR** framework, as the most promising approach to adding continuous time to this work.

3 Demonstrator Application

The demonstrator application is to be a verified real-time fault-tolerant scheduler, for a machine such as the Draper Transputer Fault-Tolerant Processor (TFTP).

Significant features of recent developments include:

- Decision to target existing TFTP hardware; this was always the probable outcome, but the availability and practicality of using the Draper hardware has been checked out.
- Characterization of control system to schedule, incorporating and refining assumptions set out in D2.1 and scoping the problem within available resource:
  - Hard Real-Time problem, with well defined data dependencies;
  - Real-world application, not dummy tasks;
  - Not more than tens of tasks;
  - Not requiring implementation of complex simulation environment interface;
  - Iteration rate within real-time performance (unscaled) of hardware.

  At a meeting in early February, a shortlist of six applications was drawn up, with a final decision (based in part on technical criteria, and in part on availability and support of original development projects) in the following quarter.
- A more precise formulation of the Byzantine agreement property, and validation of the architecture against it.
- Demonstrative application of the verification techniques previously applied to system level properties to the single-node timing proof obligations of the form generated by that higher-level analysis.

As promised, comments have been solicited from Draper on D2.5 and their feedback has been incorporated in a revised version of the document, which accompanies this report.

4 Accompanying Documents

The following documents accompany this report.

- Deliverable D1.5, in the form of a paper entitled *Specification Languages for Reactive Systems.* This introduces a range of notations suitable for describing real-time specifications and, more generally, real-time reactive systems. Topics covered include:
  - Regular expressions;
  - Temporal logics;
  - The modal μ-calculus;
  - The duration calculus;
  - Davies' approach;
  - Tabular approaches;
  - Timed CSP.

  This last gives an embedding of the operational semantics for a finitization of TCSP into the discrete "tock" model we have been using so far.
- Revised Deliverable D2.5, updated in the light of feedback from Draper.
- *Verifying Timing Properties of Static Schedulers*, a paper which expands the ideas of Working Paper W2.2.1, and provides motivating arguments for the forthcoming Deliverables D1.4 and D2.6.

References

[1] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for real-time systems. In *Proceedings Symposium on Logic in Computer Science*, pages 414–425, 1990.

[2] Rajeev Alur and D.L. Dill. Automata for Modeling Real-Time Systems. In *Proceedings of 17th ICALP*, 1990.

[3] J. Davies. Specification and Proof in Real-Time Systems. Programming Research Group Technical Monograph PRG–93, Oxford University Computing Laboratory, Oxford, England, 1991.

[4] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite-State Systems, Lecture Notes in Computer Science* **407**. Springer-Verlag, 1989.

[5] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[6] D. M. Jackson. *Logical verification of reactive software systems.* D.Phil., Oxford University, 1992.

[7] H.R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Harvard University, 1989.

[8] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of ICALP'86, LNCS 226.* Springer-Verlag, 1986. Also appears in Theoretical Computer Science 58 (1988) 249-261.

This is Release 1.0 of this document, last modified by Michael Goldsmith at 20:36:03 GMT on April 25, 1995.

FormalSystems

Specification Languages for Reactive Systems

David Jackson
Formal Systems Design & Development, Inc.

April 25, 1995

Summary

This document discusses possible additional specification and description methods to be integrated into the **FDR** system. The methods are chosen and discussed with particular relevance to real-time description in general and scheduling properties like those in [9] in particular.

We conclude with a list of required support, including suggested priorities for those to be addressed in the short term.

1 Overview

Each of the following sections discusses a method, or family of methods, which could be used to describe models or propeties of reactive systems. A wide variety of mechanisms is covered, and although all are amenable to formal description, they express a range of features and have differing needs and applications, so that we should definitely consider them as complementary, rather than exclusive.

As the desire to express properties in a more abstract form than CSP programs has been raised by a number of external agencies, we first consider methods for describing abstract constraints on behavior.

2 Regular Expressions

If we consider the whole range of computer application, possibly the greatest use of formal property expressions is to describe sequences or patterns for matching and

This is Deliverable D1.5 in the Office of Naval Research SBIR Project N00014-93-C-0213 *Embedded Transputer-based System Design.*

searching[1]. We might therefore hope to be able to take advantage of a common and unambiguous formalism taken from this area to provide an intuitive way of describing sequences of actions. This section considers using the language of *regular expressions* (regexps) to place constraints on CSP processes. Instead of matching a string or sequence of characters to a pattern, we compare a sequence of events (a trace) to a pattern expressed in terms of individual actions.

While the expressive power of regular expressions is limited, and there is no formal expressive gain over the traces model of CSP, the wide acceptability of regexps would make them an ideal engineering framework. The fact that regexps are expressible in CSP should actually greatly simplify their implementation: we may translate them into CSP process descriptions, and take advantage of the fact that the standard reduction of nondeterministic (ϵ-move) automata to determinsitic automata often used in the compilation of regexps is, in fact, a special case of CSP normalization.

A language of regular expressions for sequential behavior specifications should include at least the operations usually supported by pattern matching facilities:

Sequencing `A B` which represents a behavior matching `A` followed by one matching `B`.

Alternatives `A | B` representing behaviors which match either `A` or `B`.

Grouping using parentheses `()`.

Optional elements such as `A?` which matches either `A` or the null behavior.

Repetition either `A*` representing a behavior consisting of zero or more behaviors matching `A`, or `A+` where at least one repetition must occur.

A useful extension would be

Numbered Repetition perhaps written `n{A}m` where `n` and `m` are numbers which bound the number of repeated behaviors (each matching `A`) which constitute a behavior matching the compond regular expression.

One area where behavioral descriptions may need to differ slightly from textual applications is the specification of individual actions (corresponding to the characters in a text expression). The language should at least include

- Explicit event names
- Partially specified events (perhaps naming the channel while leaving the actual data unidentified
- Completely unspecified events (the equivalent of `.` in text expressions).

[1] Almost any word-processor includes such a feature!

The description of partially specified events will require careful consideration: the . symbol is conventionally used in CSP as a field separator, and in regular expressions as a "wildcard". It may be necessary to introduce an alternative wildcard symbol which can be combined with explicit event and data symbols using . in its CSP sense:

```
in.ANY.1
ANY
out.ANY.ANY
```

The underscore symbol _ might provide a suitable shorthand for `ANY`. We might also hope to provide a short-hand description for ranges of similar items analogous to classes of characters in text, perhaps in the following form

```
in.[0-3].[7-10]
```

Perhaps the most useful extension, however, would be support for the use of some free variables in regular expressions, to allow formal interpretation of common informal descriptions like

```
( in.X:{0..9} out!X )*
```

In the expression, of course, the usual intuitive interpretation is that the first `X` binds to a specific value, and the second refers to this. A possible alternative might be to permit a form of indexed choice: we could then write

```
( | X:0-9 @ in.X out.X)*
```

2.1 Safety properties

Actually exploiting regexps as a specification language seems possible in two distinct (and complementary) ways: we may either make positive assertions about the set of traces permitted, or we may identify specific patterns of behavior to be prohibited.

In the former case, we can restrict ourselves to a single regular expression and use the choice operations (|) to combine distinct options. We may also wish to ensure (perhaps simply by interpreting the expression appropriately) that any prefix of a permitted trace is itself permitted; it is certainly necessary for any satisfiable specfication to allow the empty trace.

The use of regular expressions to prohibit behavior patterns requires fewer restrictions, although we should still be aware that the empty trace must not be prohibited by any satisfiable specification. In this case, however, there are greater arguments for demanding that a number of separate constraints can be simulataneously imposed, as there is no straightforward operation in the usual regular expression languages which captures intersection in the way that | represents union. A variation of the latter case would allow regexps to describe system states, and then permit the user to specify

events which should not be permitted in particular states. This is a trivial variant of the second class of specifications above and may not be worth implementing directly.

In either case it is desirable to provide a means of limiting descriptions to part of a system's interface. Semantically, this should be equivalent to abstracting from other events, ideally by considering the regular expression specification interleaved with *RUN* (in the traces model) or *CHAOS* (in the failures-diveregences model) on the rest of the alphabet. Hiding irrelevant events is a less satisfactory means of abstraction as it may introduce divergence.

A more general solution would be to provide an interleaving operator in the language of expressions which could be used to perform this abstraction explicitly. This would allow the user greater freedom in choosing the patterns of abstraction, but because it is requires explicitly describing the irrelevant events, it is potentially somewhat cumbersome and confusing, so should perhaps be considered at best an alternative to a simpler scheme.

2.2 Liveness specifications

While simple regular expressions are sufficient to express safety properties, they do not allow us to specify which events *must* be possible. Perhaps the simplest way of achieving a more expressive language to allow specific refusal information, perhaps in the form of a set of events which cannot be refused, with a regular expression which determines when the condition should apply. For example we might specify a buffer as follows:

```
( in.X out.X )* OFFER in.ANY

( in.X, out.X )* in.Y OFFER out.Y
```

Note that this latter case complicates the variable scoping rules, and we need a way of specifying wildcard events in sets as well as in regexps.

If only simple assertions of this form are permitted, it should be possible to check such constraints independently. Again, however, we must provide a means of specifying which parts of a system's interface are to be constrained by a particular condition.

We should consider enriching the language of offers: conjunction and disjunction are clearly desirable, but existential and universal quantifiers might be more useful, and would permit an alternative (slightly weaker) buffer specification:

```
(in.X, out.X)*          OFFER ALL X @ in.X
(in,X, out.X)* (in.Y) OFFER SOME X @ out.X
```

2.3 Use in other contexts

In addition to the simple uses outlined here, regular expressions over traces do appear to capture many of the usual interpretations of "states" which are used in other formal specification languages. For example in the one place buffer we might use the regexp

```
( in.X out.X )*
```

to characterize the "empty" state, and

```
( in.X out.X )* in.Y
```

to charaterize "full" or "contains Y". A possible use of such state characterizations is in the duration calculus discussed in Section 5 below.

3 Temporal Logic

A wide variety of logic languages enhanced by temporal operators have been proposed. Techniques for verifying that assertions made in many of these languages hold of a range of classes of finite state machines and automata have also been widely studied. The difficulty in adopting such temporal logics for CSP specification lies in identifying a particular language which is practically applicable and consistent with the theory of CSP [8].

The larger proportion of work on applications of temporal logic to computing applications has concerned logics which refer to specific *points* in time. (A specification language based on a logic of *intervals* is discussed in § 5.) Within these point-based logics, two possible views of time are common:

- Branching time logics treat the space of all possible instants as a tree so that an instant may lead to many possible future paths, but has a single past history. This branching may be used to represent the resolution of choices as a process evolves.

- Linear time logics view the set of all instants as a total order: each instant has a single past and future ordering. In this scheme any given evolution follows a single path, but a process may be seen to behave in a variety of ways.

The most widely used branching-time temporal logic is CTL, developed by Clarke et al ([1], for example). Efficient algorithms have been developed for testing if finite-state machines satisfy CTL formulae, and the SMV model-checker has established a de-facto standard for machine-readable CTL.

The basic language of CTL couples temporal operators, such as $\diamond$ ("eventually") and $\Box$ ("forever") with quantifiers specifying how these relate to future paths. For example

Formula	Meaning
$\forall \Box \phi$	ϕ holds forever along all future paths
$\exists \Box \phi$	ϕ holds forever along some future path
$\forall \Diamond \phi$	Along all future paths, ϕ eventually holds
$\exists \Diamond \phi$	ϕ eventually holds along some future path

It should be noted that some CTL operators, particularly $\exists \Diamond$, allow us to assert that particular behaviors are possible for a system. If we consider giving meaning to such formulae in CSP, these existential specifications will assert the presence of a particular element or elements in the set of possible observations. Such specifications are inherently non-monotonic with respect to the refinement relation $\sqsubseteq$ and consequently cannot be easily used in the style of development and verification by refinement which motivates **FDR**. (In fact, these specifications may be non-constructive and thus difficult to esablish of a CSP process by any means.)

Linear time temporal logics take a somewhat different view of satisfaction and, like CSP **sat** specifications, typically insist that all possible behaviors meet a specified formula. The absence of alternative paths simplifies the form of the temporal operators, resulting in specifications like the following, for example,

$$\Box(a \Rightarrow \Diamond b)$$

which states that any a is eventually followed by a b.

The major difference between the usual style of specification in CSP and linear-time temporal logics lies in their treatment of infinite behavior. Temporal logic formulae are usually interpreted over complete views of a behavior, and the models of computation used with them assume that a process has control over its future behavior. It is thus possible to claim that a system can satisfy a specification like

$$\Diamond a$$

which asserts that a must occur eventually in any execution. The future behavior of a CSP process, however, depends on its interaction with the environment, and thus eventuality conditions must depend on assumptions about the environment's behavior. A further technical difference lies in that fact that most CSP models refer to only the finite behaviors of a process, and assume that infinite behaviors will simply be the limit of these. Temporal logic models, however, may specifically incorporate notions of *fairness* which influence only infinite behavior. A common consequence of this is that a CSP process can often only be proved to satisfy an eventuality property when it in fact satisfies a stonger condition (e.g. it may be true that eventually event b becomes available after each a, but this may be a trivial consequence of making b available as the next event after each a). For a position on fairness which matches that characterized by CSP, see [3]. Some linear-time temporal logics which attempt to avoid these problems by introducing axiomatizations of the properties of CSP behaviors are presented in [8].

4 The Modal μ-calculus

A related logic language which has attractions as a specification formalism for communication and evolution is the modal μ-calculus used by Milner, Stirling, et al. Like temporal logics, the μ-calculus uses conventional logical operators enriched with modalities which in this case take the forms

Formula	Meaning
$[a]\phi$	After any a transition, ϕ holds
$\langle a\rangle\phi$	An a transition is possible after which ϕ holds

The language also includes fixed-point operators:

Formula	Meaning
$\nu Z.\phi(Z)$	The weakest condition such that $Z \Rightarrow \phi(Z)$
$\mu Z.\phi(Z)$	The strongest condition such that $\phi(Z) \Rightarrow Z$

Additional modal operators can be defined using sets of events (e.g. after any event in set A, ϕ holds), and modal operators which ignore internal actions can also defined in terms of fixed-points.

An attractive property of the μ-calculus is that the majority of the common temporal logic operators can be encoded in it, and that it is sufficiently expressive to capture virtually all common types of properties directly (including safety, liveness and fairness). It also refers to transitions between states in a similar manner to CSP, and makes similar distinctions between visible and invisible actions. Its expressive power does itself raise some complications, however. Like branching-time temporal logics, μ-calculus specifications can make many more distinctions between processes that the theories of CSP on which **FDR** is based, including properties which are not preserved by refinement. We should expect to restrict ourselves to particular classes of formulae if we are to make optimal use of both the expressiveness of such a logical language and the power of asbtraction inherent in CSP.

5 Duration calculus

The duration calculus [12] is based on an alternative temporal logic which treats intervals, rather than points, as the basic concept (see, for example, [10]). It also provides an interpretation of properties which allows the total time for which they are true within an interval to be measured. Its intention is to provide a means of formalizing the types of property which are expressed in engineering data books by timing diagrams.

For example, the length of an interval could be expressed as the duration of the predicate *true*, written $\int true$. Then to claim that a condition *Leak* holds for no more

than one minute in every hour, we might use the predicate

$$[\int true \geqslant 3600 \Rightarrow \int Leak \leqslant 60]$$

The principle temporal operator in the duration calculus is the "chop" operator $;$. The formula $p ; q$ holds of an interval if there is some division into two adjacent intervals such that the first satisfies p and the second satisfies q.

A practical application of the calculus to CSP specification might would need two elements: the temporal element (including the usual propositional connectives) and some means of describing states (such as *Leak* above). The state descriptions should probably be based on the trace of visible events. We should probably ultimately allow both the identification of states with particular patterns (such as the regular expressions of Section 2) and with simple assertions about the values recently communicated on channels. The lack of a clearly satisfactory interpretation of state variables is currently a notable restriction on its use in conjunction with CSP.

This is a greater issue in interval-based than in point-based temporal logic because the occurrence of an instantaneous atomic communication cannot sensibly be given a duration, while such events can clearly be incorporated into a point-based logic.

Overall, the duration calculus probably offers a useful expressive power in the longer term, but its use in practice is insufficiently established to permit extensive development at this stage. It is worth noting, however, that a mechanical approach to testing the validity of duration calculus assertions has been proposed by Skakkebæk and Sestoft [11], and that this technique uses a translation into regular expressions of the sort described above, placing further emphasis on the importance of that technology.

6 Davies-style Macros

One approach which has been proposed specifically for specifications using the Timed CSP model is "macro" language for Timed-Failure specification proposed by Davies in [2]. In essence, this language is a first-order prodicate logic with variables ranging over time values. Assertions may be made about the availability or occurrence of events at specified times:

Formula	Meaning
a at t	Event a is observed at time t
a from t	Event a is available from time t until it occurs
a from t until t'	Event a is available from time t until it occurs or time t' is reached
a from t until b	Event a is available from time t until either a or b occurs

These expressions can be extended to allow non-determinism by permitting sets of events or times to be specified. To support concise specification of processes involving the hiding operator, Davies also defines an "active" predicate which asserts that the environment never prevents actions on a given channel. The language can be extended to include functions on the sequences of actions performed during specified intervals, including, for example,

- the sequence of data on a specific channel,
- the number of occurrences of events in a given set, and,
- the time of the last occurrence of an event in an interval.

An example of this style of specification can be found in [5, Section 7].

Using a general predicate logic as the underlying logical framework gives Davies specification style a great deal of expressive power, but does not lead easily to mechanical verification. Perhaps the most attractive approch is to consider properties of behaviors expressed in the style as the "state" formulae in a temporal or modal language. For example, a valve controlled by *open* and *close* events may be characterized by two states expressed as predicates on the timed trace s:

$$\begin{aligned} \mathit{Open} &\;\widehat{=}\; \text{last}\,(s \restriction \{\mathit{open}, \mathit{close}\}) = \mathit{open} \\ \mathit{Shut} &\;\widehat{=}\; \text{last}\,(s \restriction \{\mathit{open}, \mathit{close}\}) \neq \mathit{open} \end{aligned}$$

We might then insist that over any 10 minute period the value was shut more often than it was open by a duration-calculus style formula:

$$\Box(\ell \geqslant 600 \Rightarrow \textstyle\int \mathit{Shut} \geqslant \int \mathit{Open})$$

Based on the standard models of *Timed* CSP, this approach conventionally makes reference to refusal (i.e. liveness) information throughout the whole evolution of a process. Although the current mappings of timed analysis into the untimed domain do not entirely match this model, the prioritization operator of [4] does allow us to express similar properties to those which would be expressed in Timed CSP by a refusal predicate. To express the notion that an event a always occurs when offered by a process in Timed CSP, we take as an hypothesis that the event is *active*: a is perceived (by the environment) to be refused at all times. A similar concept can be encoded in a prioritized model of untimed CSP by giving a a high priority: time (represented by a lower priority event) will only pass when a cannot occur. We should also note that *stable* refusals may be constrained in a similar manner in both Timed and untimed CSP: in both frameworks we may insist, for example, that a is possible following a b event by claiming that when the *latest* event is b, a should not be stably refused. Because TCSP behavior sets are prefix-closed, this is sufficient to guarantee that a is available at appropriate times throughout an execution, even though our untimed assertion considers only behaviors ending in b.

7 Tabulated Functions

The preceding sections have discussed a variety of methods of describign specific properties of systems. The next two describe some description techniques which experience on this project have indicated would be useful in describing the models rather than their properties.

One of the most potentially useful vehicles for requirements capture does not address the definitions of processes or models at all, but simply reduces the effort required to create and maintain models: the ability to capture tabulated or structured data easily is a great asset.

We require two features of a data capture facility:

- It should take input in a standard form or forms, e.g. from a common spread-sheet format, and
- The captured information should be easily accessible within a model

The choice of preferred data source is in fact relatively straightforward: apart from propietary formats, spread-sheet packages share few common interchange formats, of which the Comma Separated Value (CSV) form is most popular. We propose that **FDR** be extended to allow CSV files to be used to define model parameters.

Providing an interface to this data within a model requires further clarification, however. Perhaps the cleanest form of integration would be to allow function definitions to be tabulated:

```
pragma tabulated f, g

P(x) = a ! f(x) -> Q(g(x))
```

The `pragma` declaration must include the name of the function, and possibly also

- the name of the file containing the definition,
- the number and type of the arguments to the function, and,
- the type of the result.

As well as simple evaluation of functions, the CSP interface to a data table should include some way of determining the domain of a tabulated function, both as a true set and also as a sequence which reflects the order in the original source. These simple facilities would be sufficient to support practical applications such as allowing the Engine Management System model of [9] to be maintained by engineers without detailed knowledge of CSP.

To facilitate this type of usage, the **FDR** facility should permit

- Functions of up to at least three arguments
- Multiple functions defined over the same domain in a single table
- Interrogation of a functions domain as a set and a sequence

A further development would be to allow additional information to be included in a spread-sheet format, including, say, definition files to be loaded and checks to be performed. This would allow a suitably configured **FDR** model to be operated entirely from a non-CSP data source.

7.1 State transition tables

A natural side effect of providing this facility (which could perhaps be further exploited by providing a suitable library function) allows state machines to be constructed from state-tables in a fairly simple manner. If we define a function `next(in,curr)` which relates input events and current state values to successor states, the simple CSP process

```
P(s) = [] x : alphaP @ x -> P(next(x,c))
```

provides an animation of the state machine defined by `next`. This trivial model can of course be extended to restrict permitted inputs and to allow outputs to be generated in the usual Moore machine style, as in the following example which uses the following functions

`inputs(s)` Returns the set of inputs expected in state `s`

`outputs(s)` Returns the set of outputs possible in state `s`

`next(s,v)` Defines the state reached from `s` when `v` is communicated.

```
P(s) = ([] x : inputs(s) @ x -> P(next(s,x)))
       |~|
       (|~| y : outputs(s) @ y -> P(next(s,y)))
```

The transition function `next` can be expressed as a function of two variables, either in a simple tabular form:

Current	Input	Next
1	a	2
1	b	3
1	c	4
2	b	1
2	c	6
⋮		

or as a two-dimensional grid:

Current	Input a	b	c	
1	2	3	4	
2		1	6	
				⋱

In wider use, it would be desirable to allow a wider range of constructs to provide fuller support for state variables and channel values. One approach would be to allow a set of variables to be defined for each state-transition table. These could be referenced in the column headings of a state transition matrix by the ? and ! notation:

```
TABLE copy
VARIABLE x
INITIALLY empty
TRANSITIONS
        ,   in ? x   ,   out ! x
empty   ,   full     ,
full    ,            ,   empty
```

(Either numbers, or preferrably, symbolic names could be used in the vertical axis of such a table as state identifiers).

An alternative scheme (but one which is possibly over complex) is to associate variables with states. In defining the "next" function, event labels with ? then bind that name in the successor state, event labels containing ! use the value in the current state.

	in ? x	out ! x
empty	full(x)	
full(x)		empty

The direct translation from either of these forms of table to CSP is straightforward.

8 Timed CSP

Given our work on discrete time, an obvious enhancement to the modelling capabilities of FDR is to extend the input language to support *Timed* CSP and allow its translation to a clocked process.

The additional operators which need to be supported are

WAIT t Delay

$a \xrightarrow{t} P$ Timed prefixing

$P \lightning\{t\}\ Q$ Timed interrupt

$P \rhd\{t\}\ Q$ Time-out choice

In addition, fuller support for the untimed CSP interrupt operators is desirable, including

$P \triangle Q$ General interrupt

Support for older, less general, interrupt operators would be advantageous, and could be provided by syntactic transformation:

$P \underset{a}{\triangledown} Q$ Event-triggered interrupt

$$P \underset{a}{\triangledown} Q \quad = \quad P \triangle a \rightarrow Q$$

$intr(a, P)$ Resetting interrupt,

$$intr(a, P) \quad = \quad \mu X \bullet P \triangle a \rightarrow X$$

The required semantics of the simpler processes in a discrete time framework are quite clear; we suggest the following, where χ is the distinguished event which represents the passage of a unit of time, and $[\![P]\!]$ represents the discrete-time translation of P.

$$[\![WAIT t]\!] \quad \widehat{=} \quad \text{if } t = 0 \text{ then } SKIP \text{ else } \chi \rightarrow [\![WAIT(t-1)]\!]$$

$$\begin{aligned} [\![a \xrightarrow{t} P]\!] \quad &\widehat{=} \quad WAITING(a, t, P) \\ WAITING(a, t, P) \quad &= \quad \chi \rightarrow WAITING(a, t, P) \\ &\quad \Box \\ &\quad a \rightarrow [\![WAIT t]\!] \,;\, P \end{aligned}$$

$$\frac{P \xrightarrow{a} P'}{P \lightning\{t\}\ Q \xrightarrow{a} P' \lightning\{t\}\ Q} \quad [\, a \neq \chi \,]$$

$$\frac{P \xrightarrow{\chi} P'}{P \lightning\{t\}\ Q \xrightarrow{\chi} P' \lightning\{t-1\}\ Q} \quad [\, t > 0 \,]$$

$$\frac{}{P \not\downarrow \{0\}\ Q \xrightarrow{\tau} Q}$$

$$\frac{P \xrightarrow{\tau} P'}{P \rhd \{t\}\ Q \xrightarrow{\tau} P' \rhd \{t\}\ Q}$$

$$\frac{P \xrightarrow{a} P'}{P \rhd \{t\}\ Q \xrightarrow{a} P'} \quad [\ a \notin \{\chi, \tau\}\ ]$$

$$\frac{P \xrightarrow{\chi} P'}{P \rhd \{t\}\ Q \xrightarrow{\chi} P' \rhd \{t-1\}\ Q} \quad [\ t > 0\ ]$$

$$\frac{}{P \rhd \{0\}\ Q \xrightarrow{\tau} Q}$$

For the interrupt operator:

$$\frac{P \xrightarrow{a} P'}{P \triangle Q \xrightarrow{a} P' \triangle Q} \quad [\ a \notin \mathit{initials}(Q)\ ]$$

$$\frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'} \quad [\ a \neq \tau\ ]$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'}$$

Practically, there are two possible approaches to integrating these operators with untimed CSP: we might construct separate definitions in a purely timed language and provide a global translation, but it seems perhaps preferable to allow a mixture of processes definitions using the above operators (where timing information is implicit in the syntax) and untimed processes, or at least processes in which the timing events are explicit. In adopting the latter course, it will be necessary

- to ensure that the semantics of untimed processes are consistent with the standard models, or at least can be made to be so by user-selectable option.
- to allow the additional features to be disabled in such a way as to prevent accidental usage in conventional CSP scripts

A fuller interface to prioritized checks, and other functions or operators particularly useful in a timed context is obviously necessary in the longer term.

9 Quantitative Results

Reference [9] shows that numerical information may be extracted from CSP models by including families of additional transitions and recording those which are actually enabled in reachable states of the combined specification and implementation machines. While this approach is still in the early phases of development, we may at least consider how a more mature technology could be integrated into our system. One possibity is to introduce annotations into the CSP model of a specification which make assignments to measured parameters when specific *transitions* are taken. For example

```
Deadline(x) = tock -> Deadline(x-1) [] done {margin := x} -> Reset
```

An alternative approach could associate parameters with the states such as `x` in `Deadline(x)` above.

10 Summary: Required Functionality

We propose that the following extensions to **FDR** be provided in the short term:

- Tabulated functions and simple state-transition tables
- Regular expression descriptions of safety properties
- Discrete-time interpretations of TCSP operators

That within the life of the current project, the functionality should be extended to include:

- Regular expression descriptions of liveness properties
- Support for a modal- or temporal- logic (perhaps the Duration Calculus) by property checking in some form, at least covering safety properties,

and that the following specification formalisms are considered for longer term support:

- timing specific specification languages in the Davies style, possibly coupled with
- fuller support for temporal logic property descriptions

References

[1] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming languages and Systems*, 8(2):244–263, 1986.

[2] J. Davies. Specification and Proof in Real-Time Systems. Programming Research Group Technical Monograph PRG–93, Oxford University Computing Laboratory, Oxford, England, 1991.

[3] E.W. Dijkstra. Position paper on "fairness". *Software Engineering Notes*, 13(2):18–20, April 1988.

[4] M.H. Goldsmith. A CSP Priority Operator for **FDR 2**; Prototype Software for Discrete Real-time Extensions to **FDR**. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [6].

[5] M.H. Goldsmith. Embedded transputer-based system design: Final report. Report on ONR SBIR contract N00014–91–C–0054, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1995.

[6] M.H. Goldsmith et al. *N00014-93-C-0213: Fourth Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[7] M.H. Goldsmith et al. *N00014-93-C-0213: Fifth Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1995.

[8] D. M. Jackson. *Logical verification of reactive software systems.* D.Phil., Oxford University, 1992.

[9] David M. Jackson. Verifying Timing Properties of Static Schedulers. Technical report, Formal Systems Design & Development, Inc., 1995. Deliverable to SBIR N00014-93-C-0213, in [7].

[10] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, The Pitt Building, Trumpington St, Cambridge, UK, 1986.

[11] J. U. Skakkebæk and P. Sestoft. Checking validity of duration calculus formulas. ESPRIT BRA 7071 (ProCoS) Project Report [ID/DTH JUS 3/1], Technical University of Denmark, 1994. Available as `ftp://ftp.id.dth.dk/pub/ProCoS/Jens.U.Skakkebaek/IDDTH-JUS-3-1.ps.Z`.

[12] Zhou ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. Technical Report OU ZCC2, ESPRIT ProCoS project, 1991.

This is Release 2.0 of this document, last modified by Michael Goldsmith at 20:34:42 GMT on April 25, 1995.

FormalSystems

Models of the Fault-Tolerant Processor Architecture and Verification

David Jackson
Richard Chapman

April 25, 1995

Summary

This document describes our work to date on formalizing the design and analysis of the Transputer Fault-Tolerant Processor system. The early sections summarize the fault-tolerance properties which we intend to verify, our model and a simple demonstration that the architecture does meet our requirement for Byzantine fault-tolerance. We then describe how such verification can be simplified if we exploit the symmetry both the overall design, and of the behavior of its components. The next section describes how we can relate these models of network behavior to the application level scheduling problem, and in particular how we can exploit temporal redundancy to tolerate transient faults. It includes discussions on voting, permuted schedules and on transient recovery techniques. Abstract models of task execution and voting are given which, despite their simplicity, provide a framework for future models of specific scheduling policies. We include two more detailed models of the system which analyse a distributed model of the system using synchronous and partly asynchronous models.

This is a revised Deliverable D2.5 in the ONR SBIR project N00014–93–C–0213 *Embedded Transputer-based System Design.*

Contents

1 Fault-Tolerant Behavior

We begin by summarizing the types of behavior which we ultimately intend to analyse in order to show how they can be expressed in terms of the models we will describe below.

1.1 Classes of Fault-Tolerance

The first class of faults we will consider, and the errors they may cause are, those outlined in a previous project document, [2]. These are the *Byzantine* failures of

a system element, after which no assumptions can be made about the behavior of the component. In particular, such faults may not be manifest – failure may not be obvious to those parts of the system with which the failed element interacts. It is well-established that a suitably redundant system can tolerate Byzantine failures, but that the cost of such a system is higher (i.e. it requires greater redundancy) than a system designed to tolerate manifest faults (see, for example, [10]).

We can exploit this redundancy, however, to improve tolerance to other types of fault. Of particular importance are "common-mode" errors which arise in a number of replicated elements simultaneously, perhaps as the result of some environmental factor. Where these errors arise from transient faults (such as corruption of semiconductor memory) we can use *temporal* redundancy to allow correct operation to be resumed even if the number of elements affected is much greater than the number of Byzantine failures that a system might tolerate. This strategy has again been outlined in a previous project report [1].

Obviously these two situations are far from being an exhaustive catalogue of fault situations which we might design a system to tolerate but they do represent a possible extremes: in the Byzantine case we suffer complete non-manifest failure of few components, in the transient case we tolerate identifiable temporary faults in many. Other combinations, such as manifest permanent faults, may be included in later analysis.

1.2 Fault models

Modeling a component capable of Byzantine failure is relatively straight-forward, because we need to satisfy very few constraints on behavior after an error, but we must nevertheless take into account the features that our model represents if we are to provide a satisfactory model.

High level abstractions In models at the highest level of abstraction (the replicated synchronous view of [2]), a failed component can be represented as ignoring all inputs. We choose to ignore, rather than to refuse, inputs in order to remove the need to model details of the error detection and buffering which is used in practice to implement communication between distributed components. These communication elements are, of course, modeled in the lower level abstractions (Section 6 of this report). Abstracting from the implementation of the communication and error detection mechanism also influences the way we should model outputs from a faulty system element. The most obvious approach is to allow arbitrary generation or refusal of output events. This correctly captures the idea that a failed component exhibits the most general possible behavior but does not reflect the ability of a receiver to detect when outputs are being refused (typically by means of a time-out). We therefore model a faulty output as a combination of arbitrary valid outputs together with

a distinguished error value which is always potentially available. While placing constraints on faulty behavior may appear unrealistic, it should be remembered that our fault model is actually also incorporating a significant amount of information about the ability of connected components to detect faults. We will make this information more explicit in later sections, but this abstract model will remain useful *because* it does not make assumptions about how communication errors are detected, and thus applies to a wide range of possible error detection techniques, including time-outs, parity or check-sum errors or more complex protocols.

Lower level abstractions In more detailed models the models of faulty components actually become simpler, because we are able to model more faithfully the way in which errors are detected by the remainder of the system. Both input and output behavior of a process after the occurrence of a Byzantine fault can be assumed to be entirely arbitrary: both inputs and outputs can be performed in any order, or refused at any stage. This is exactly the behavior of the $CHAOS$ process of CSP, as we might expect of a completely undetermined behavior.

By their very nature, transient faults require a more detailed model of the internal state of a system than Byzantine failure. The essence of our approach will be to decompose the application calculations into a series of tasks each of which calculates new values for part of the system state (and may produce outputs) from the previous system state and any inputs present. A transient fault is modeled by assuming that the fault corrupts some part of the processor's state arbitrarily, and that all tasks depending on that part of the state may in turn corrupt their outputs and final states. The task of the fault management system is to identify the corrupted parts of the system state and re-generate it where possible. Adding sufficient information to our high-level model to support this reasoning is discussed in Section 4 and later sections.

2 High-level Architecture

For practical applications, we will assume that tolerance of a single Byzantine fault is sufficient, and thus we will concentrate on quad-redundant systems. Each of the four redundant *fault-containment regions* (FCRs) which make up such a system must execute both the application tasks and the functions related to fault management: in our demonstrator application each FCR will typically contain two processors, one executing the application and another managing communication and input-output. This bipartite view is also applicable to single processor systems built using Transputer hardware, as separation between processing and communication is present even if the components are actually a CPU and a link engine on a single IC.

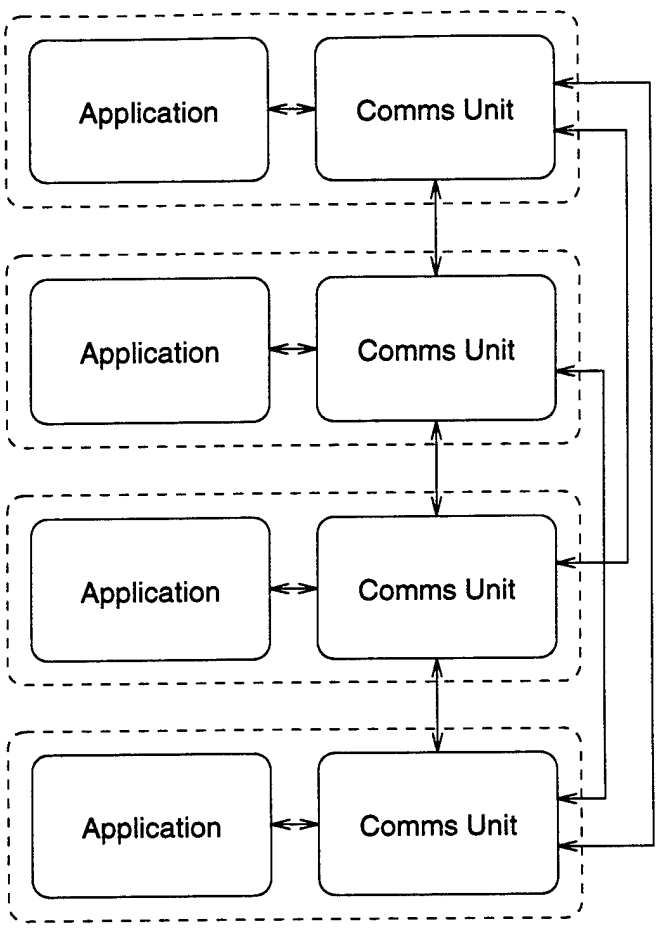


Figure 1: Communication between peers

2.1 Implementing the Oral Messages algorithm

As discussed in [2], we will use the *Oral Message* (OM) algorithm to establish consensus values for data in the presence of faults. Each FCR will communicate its local values for state and output data to its peers, and vote upon a derived value using its local data and the values it receives in return. The communication will have the pattern shown in Figure 1. Each node in Figure 1 represents the communications processing element of an FCR. Data is received from the application along the *in* channel and passed out along the cross channels (the vertical links in the diagram). Values received, along with the original value received, are combined by a majority voting process and the result is passed to the application or the environment.

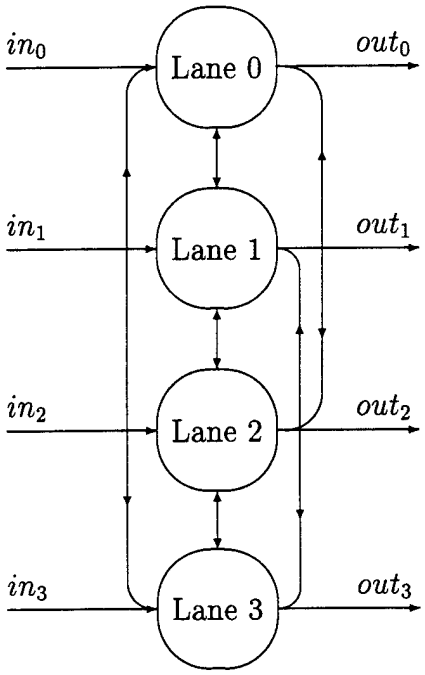


Figure 2: High-level Model Architecture

3 Tolerance of Byzantine Faults

We start our analysis with a high-level abstraction which serves to justify our primary claim of tolerance to Byzantine faults. The following model has the same structure as that outlined in Section 2.1. We concentrate on modeling the communication behavior of the system, and thus model the behavior of the input-output subsystem alone, representing data flowing to or from the application or IO devices by sets of channels *in* and *out*. The pattern of communication is then as shown in Figure 2. The channels *in* and *out* may not, of course, exist as explicit data paths in the case that application processing and communications are combined on a single processor, but there will always be some identifiable transfer of data corresponding to them. Each FCR (i.e. each node in Figure 2) is represented by two processes, one representing the outward transfer of local data to peer FCRs, the other representing voting using data received.

The desired behavior of our system is described in [2] in terms of two properties of a system distributing data from a single source by means of a two-stage algorithm. The properties are:

Agreement If two processors are non-faulty, they agree on the data values which they believe are being communicated.

Validity If the originator of a data item is non-faulty, all non-faulty processors derive the correct value.

To model the two stage transmission we will consider a network consisting of the four communication elements of our system together with an addition process which performs the initial data distribution. In a physical system we would expect this additional task to be implemented within one particular FCR, the *transmitter* of the data flow being considered, while the other FCRs would be *receivers* of the flow.

We can verify a variety of properties of the system by adding components to our network which do not correspond to any actual implementation processes, but rather capture our ability to observe the system. For example, one reasonable property which captures some aspects of the validity condition (although it is strictly weaker than the version given above) is the claim that a majority of the outputs of all channels should agree on the correct value, as long as the transmitter is functioning. We may demonstrate that our system satisfies this requirement by adding a final overall majority voting process to the system. If each FCR delivers the value it computes to this final vote, then if the validity property holds of outputs of the FCRs, the output of the voter must always match the value provided by the data source. The overall data-flow through the network is shown in Figure 3. We require that this complete system, when viewed as a data transmission medium between its source and the final output, is a perfect buffer, provided that the first-round data distribution is non-faulty. This must hold even if one of the receiver FCRs is Byzantine faulty. A CSP model of this system (suitable for analysis with the **FDR** [3] tool) is given below.

tftp.csp: Model demonstrating tolerance of 4-FCR Oral-Messages algorithm to a single Byzantine fault.

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Dave Jackson.
`-- This version: $Id: tftp.csp,v 2.2 1995/04/20 20:45:12 dave Exp $`

In the current model we are principally interested in the distinction between a real data value and a potentially erroneous one. It will suffice, for the present, to consider a single "good" data value, and an error token, Err:

`RAWDATA = {0}`

`Err = 99` *Any value not in RAWDATA*

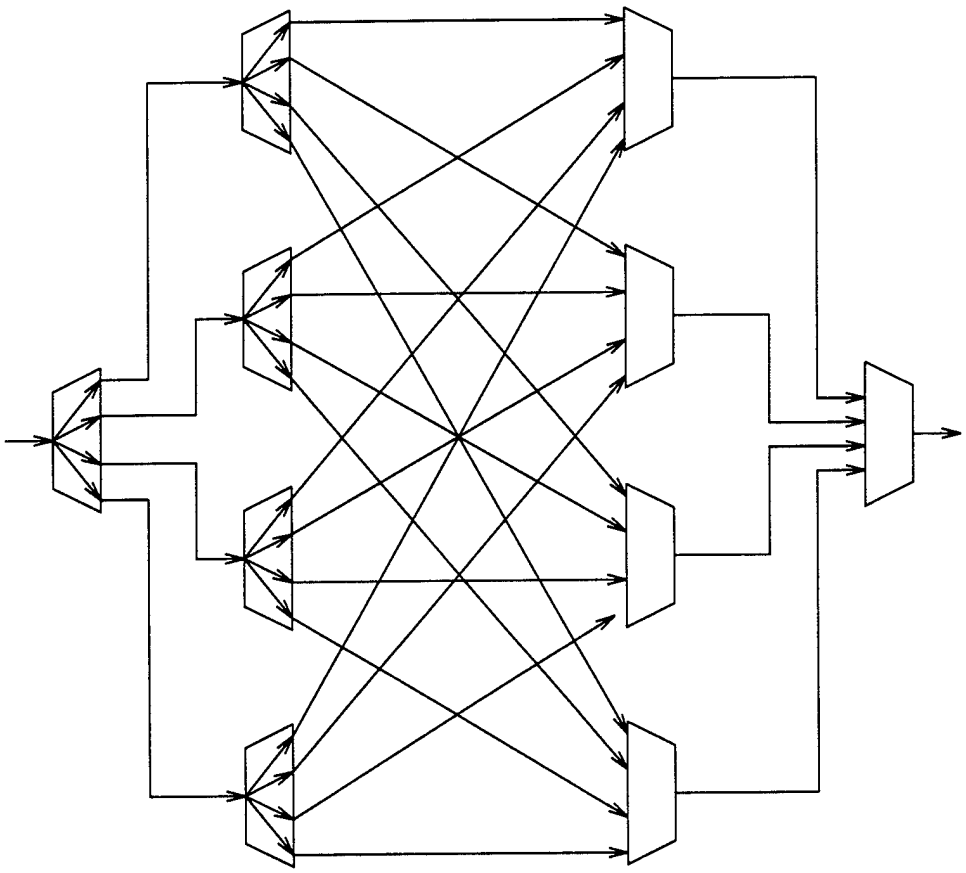


Figure 3: Detail of Data-flow Through FTP Model

```
DATA = union({Err},RAWDATA)
```

Specification of data exchange mechanism

We require that data is transmitted from the data source to the outputs of each communications element in such a way that a majority vote over all those outputs correctly reflects the input. Our high level specification is thus that the system is a buffer. We can in fact show that the system represents a deterministic buffer, as follows:

The initial state of an n place buffer is empty:

```
BUFFER(n) = BUFF(<>,n)
```

For any state of the buffer, if it is empty it must accept an input.

```
BUFF(t,n) =
  if (null(t)) then source?x:RAWDATA -> BUFF(<x>, n)
  else
```

Otherwise, if the buffer is full it offers only an output.

```
if ((#(t))==n) then sink!(head(t)) -> BUFF(tail(t), n)
```

The final case, where the buffer is neither full nor empty allows both input and output.

```
else            (source?x:RAWDATA -> BUFF(t^<x>, n))
                    []
                     (sink!head(t) -> BUFF(tail(t), n))
```

At the time of writing, the FDR tool requires that we specify a fixed maximum size for our specification:

```
BUFF4 = BUFFER(4)
```

(This restriction is not theoretically necessary and we expect to be able to relax this constraint in future versions of FDR2.)

Model of OM Algorithm for Four FCRs

The most complex basic component in the algorithm is the voting module: the following process takes inputs from the channels specified in the set sources and passes majority voted values to the channel sink

Voting is encoded by maintaining sets of those channels which have supplied values for each data type, including error returns. Initially these sets are empty:

```
MAJ(sources, sink) = MAJORITY(sources, {}, {}, {}, sink)
```

While it is accepting input, the voter offers a choice over the inputs which it is still expecting to receive. When input is received, the channel is added to the appropriate set, and removed from the set of exepected inputs.

```
MAJORITY(expected, zeroes, ones, errs, sink) =
  if (card(expected) == 0) then OUTPUT(zeroes, ones, errs, sink)
    else ([] x : expected @ (x?y ->
      (if (y==0)
        then MAJORITY(diff(expected,{x}),
                      union(zeroes,{x}), ones, errs, sink)
        else if (y==1) then
             MAJORITY(diff(expected,{x}),
                      zeroes, union(ones,{x}),errs,sink)
        else
             MAJORITY(diff(expected,{x}),
                      zeroes, ones,union(errs,{x}),sink))))
```

When all expected inputs have been received, the voter supplies an output according to the size of the sets of inputs received. (For a single element data domain, we output the same (valid) value for any combination of valid inputs.)

```
OUTPUT(zeroes, ones, errs, sink) =
  if ((card(zeroes)==4) or (card(zeroes)==3))
    then sink!0 -> MAJ(Union({zeroes, ones, errs}), sink)
    else sink!Err -> MAJ(Union({zeroes, ones, errs}), sink)
```

The other required component is a data distribution process. While we could write this in a sequential form similar to the voter, we feel the symmetry of the action is made clearer if we express this process as a parallel composition of simple buffers. These buffers synchronize on their input but not on their output, yielding the required interleaving behaviour.

```
COPY = inp ? x:RAWDATA -> o0 ! x -> COPY
```

The following channel definitions specify the input to, and outputs from the first-stage data distribution. Later instances of the data distribution process will be derived by renaming this first one:

```
pragma channel inp : DATA
```

```
pragma channel o0, o1, o2, o3 : DATA

INSERT =  COPY [| {inp } |]
         ((COPY[[ o0 <- o1 ]]) [| { inp } |]
         ((COPY[[ o0 <- o2 ]]) [| { inp } |]
         (COPY[[ o0 <- o3 ]])))
```

The following channels define the external interfaces to our model:

```
pragma channel source, sink : DATA
```

and these implement the connections between peers:

```
pragma channel xcmid : DATA
pragma channel xc01, xc02, xc03 : DATA
pragma channel xc10, xc12, xc13 : DATA
pragma channel xc20, xc21, xc23 : DATA
pragma channel xc30, xc31, xc32 : DATA
```

and finally, the channels which represent the input and output from each of the FCRs:

```
pragma channel ain, bin, cin, din : DATA
pragma channel aout, bout, cout, dout : DATA
```

For brevity in later descriptions, we define sets of channels representing the inputs:

```
XCI0 = {xc10, xc20, xc30}
XCI1 = {xc01, xc21, xc31}
XCI2 = {xc02, xc12, xc32}
XCI3 = {xc03, xc13, xc23}
```

and outputs

```
XCO0 = {xc01, xc02, xc03}
XCO1 = {xc10, xc12, xc13}
XCO2 = {xc20, xc21, xc23}
XCO3 = {xc30, xc31, xc32}
```

connecting each FCR to its peers. The total interface sets of each FCR are as follows:

```
ALPHAA = (Union({{ain, aout}, XCI0, XCO0}))
ALPHAB = (Union({{bin, bout}, XCI1, XCO1}))
ALPHAC = (Union({{cin, cout}, XCI2, XCO2}))
ALPHAD = (Union({{din, dout}, XCI3, XCO3}))
```

We may now define processes representing each FCR. Each consists of a data distributor communicating with a voter by a channel xcmid. The data distributor also provides outputs $XCOn$ and the voter accepts inputs from set $XCIn$.

```
FTLANEA =
  ((INSERT [[inp<-ain, o0<-xcmid, o1<-xc01, o2<-xc02, o3<-xc03]])
  [(union({ain, xcmid}, XCO0))||(union({aout, xcmid}, XCI0))]
  (MAJ(union(XCI0,{xcmid}), aout)))
  \ {xcmid}

FTLANEB =
  ((INSERT [[inp<-bin, o0<-xcmid, o1<-xc10, o2<-xc12, o3<-xc13]])
  [(union({bin, xcmid}, XCO1))||(union({bout, xcmid}, XCI1))]
  (MAJ(union(XCI1,{xcmid}), bout)))
  \ {xcmid}

FTLANEC =
  ((INSERT [[inp<-cin, o0<-xcmid, o1<-xc20, o2<-xc21, o3<-xc23]])
  [(union({cin, xcmid}, XCO2))||(union({cout, xcmid}, XCI2))]
  (MAJ(union(XCI2,{xcmid}), cout)))
  \ {xcmid}

FTLANED =
  ((INSERT [[inp<-din, o0<-xcmid, o1<-xc30, o2<-xc31, o3<-xc32]])
  [(union({din, xcmid}, XCO3))||(union({dout, xcmid}, XCI3))]
  (MAJ(union(XCI3,{xcmid}), dout)))
  \ {xcmid}
```

The fault-tolerant communication system as a whole is a parallel combination of these:

```
FTBUFF =
  (((FTLANEA [ALPHAA||ALPHAB] FTLANEB)
  [union(ALPHAA, ALPHAB)||union(ALPHAC,ALPHAD)]
    (FTLANEC [ALPHAC||ALPHAD] FTLANED)) \
                Union({XCI0, XCI1, XCI2, XCI3}))
```

The following sets define the interfaces of the first-level data distribution, the voting network just defined, and the majority voter used to complete the model.

```
ALPHAIN = {source, ain, bin, cin, din}
ALPHAFT = {ain, bin, cin, din, aout, bout, cout, dout}
ALPHAMJ = {sink, aout, bout, cout, dout}
```

These components are combined as follows:

```
SYSTEM =
  (((INSERT [[inp<-source, o0<-ain, o1<-bin, o2<-cin, o3<-din]])
  [ALPHAIN||ALPHAFT]
  FTBUFF)
  [(Union({ALPHAIN, ALPHAFT}))||ALPHAMJ]
  MAJ({aout, bout, cout, dout}, sink)) \ ALPHAFT
```

We hope, and indeed find, that $BUFF4 \sqsubseteq SYSTEM$

Now consider a failed processor, assumed not to be the source of single source data:

```
RUN(A) = [] a:A @ a -> RUN(A)

FTLANED' = RUN(Union({events(i)| i<- union({din},XCI3)}))
           ||| CHAOS(Union({events(i) | i <- union({|dout|},XCO3)}))
           ||| RUN({c.Err | c <- union({dout},XCO3)})
```

NB broken channel always allows error outputs.

```
FTBUFF' =
  (((FTLANEA [ALPHAA||ALPHAB] FTLANEB)
  [union(ALPHAA, ALPHAB)||union(ALPHAC,ALPHAD)]
    (FTLANEC [ALPHAC||ALPHAD] FTLANED')) \
                Union({XCI0, XCI1, XCI2, XCI3}))

SYSTEM' =
  (((INSERT [[inp<-source, o0<-ain, o1<-bin, o2<-cin, o3<-din]])
  [ALPHAIN||ALPHAFT]
  FTBUFF')
  [(Union({ALPHAIN, ALPHAFT}))||ALPHAMJ]
  MAJ({aout, bout, cout, dout}, sink)) \ ALPHAFT
```

We observe (for a single element data set + error value) that $BUFF4 \sqsubseteq SYSTEM'$ *(approx 2.4M state pairs).*

For a straightforward agreement, we use a simpler final specification

```
pragma channel error

AGREE(sources) = AGR(sources, {}, {}, {})
```

While it is accepting input, the voter offers a choice over the inputs which it is still expecting to receive. When input is received, the channel is added to the appropriate set, and removed from the set of exepected inputs.

```
AGR(expected, zeroes, ones, errs) =
  if (card(expected) == 0) then AGRTEST(zeroes, ones, errs)
    else ([] x : expected @ (x?y ->
      (if (y==0)
        then AGR(diff(expected,{x}),
                 union(zeroes,{x}), ones, errs)
        else if (y==1) then
             AGR(diff(expected,{x}),
                 zeroes, union(ones,{x}),errs)
        else
             AGR(diff(expected,{x}),
                 zeroes, ones,union(errs,{x})))))

AGRTEST(zs,os,es) =
        if card(zs) >= 3 or card(os) >= 3 then
                AGREE(Union({zs,os,es}))
        else error -> STOP


SYSAGREE =
  (((INSERT [[inp<-source, o0<-ain, o1<-bin, o2<-cin, o3<-din]])
  [ALPHAIN||ALPHAFT]
  FTBUFF')
  [(Union({ALPHAIN, ALPHAFT}))||ALPHAMJ]
  AGREE({aout, bout, cout, dout})) \ ALPHAFT

SPECAGREE = CHAOS({|source,sink|})
```

Similar models can be used to verify validity directly by replacing the voter with a specific observer component. If the observer signals an error whenever the valid outputs do not calculate the correct value, we may demand that no such errors are never signalled, and so establish validity. Agreement can be specified in this style.

4 Refining the Architecture

4.1 Exploiting Symmetry

Whether verification is carried out by hand or mechanically, analysis of detailed models of the fault-tolerant processor will involve significant effort. From the structure of the model, however, we can see a very clear symmetry between the four components of the network.

We can exploit this symmetry in a number of ways, the most obvious being a reduction in the number of possible failures to be considered. If the network is operating in a fully symmetric manner, it obviously does not matter which of the processors is considered to fail, and we may thus isolate failures to an arbitrary fixed FCR. If the operation is asymmetric, as in the distribution of single-source data, we way still exploit the three-fold symmetry of the receiver processes and model single faults by only two cases: either the transmitter fails, or one of the receivers does. In the latter case the identity of the failed lane can again be arbitrary.

A more powerful technique exploits not only the large-scale symmetry of the network, but also takes advantage of the symmetric behavior of the components. Suppose, for example, that the data distribution process is entirely symmetric as regards the order and manner in which it makes its output available, as is the case for the process *INSERT* in the model of Section 3. Recall the data-flow shown in Figure 3; if we concentrate on the values produced by any one of the local voters, we see that it depends on only four of the data interchanges (i.e. the value from its local input and values received from its three peers. The relevant paths are highlighted in Figure 4. The behavior of each data distribution phase when we consider only one of its outputs will be significantly simplified, and indeed in many cases[1] it will degenerate to a simple form of buffer. The overall system which we must analyse to predict the output of a given voter reduces to that shown in Figure 5. We can use this simplification to allow us to prove properties of the whole system by considering only a single voter. Suppose we show that the output of the voter in the figure agrees with its input provided that no more than one of the preceding buffers (representing the data distribution operation of each FCR) is faulty. Unless we use explicit assumptions about which voter we consider and which buffer is faulty, our reasoning must then

[1] Typically those where blocking one output does not prevent further inputs and outputs on other channels.

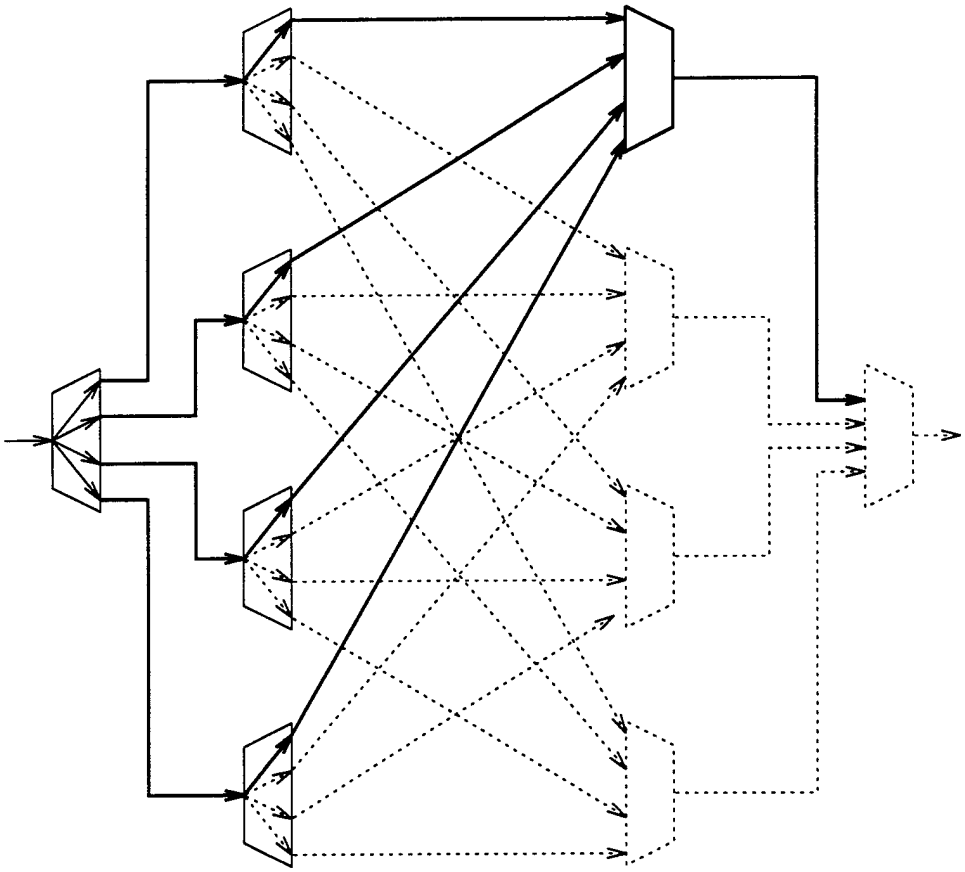


Figure 4: Data-flow to a Single Voter

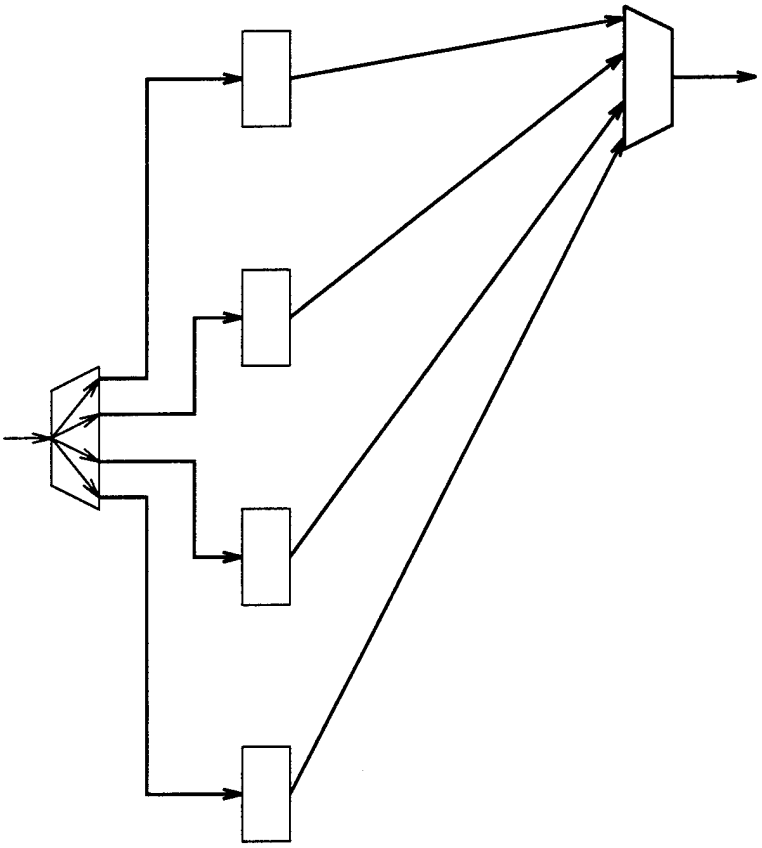


Figure 5: Analysing the Output of a Single Voter

be valid for any functioning voter, and any single failure: we have established that *all* functioning voters agree with the input. In practice, of course, not all the data distributors will be identical in their relationship to the voter, because one of them will be contained in the same FCR, and may be implemented by the same processor. However, presence of a Byzantine fault in this distributor will then imply a potential fault in the voter, and we do not need to (and cannot expect to) establish that FCRs behavior.

This approach allows our models to concentrate on the behavior of a single component in our system, rather than having to model and analyse four identical replicas. We shall use this technique to demonstrate agreement propeties of the system in the following section.

4.2 Verifying the Agreement Property

In Section 3, we commenced our analysis with the system's *validity* property, as being a more practically motivated and in some respects stronger condition than the agreement property. (As all functioning voters yield the value provided by the input, it is obvious that all such nodes do agree!) The basic agreement property is still important, however, as it gives us assurance of consistency between receivers, even if a data source becomes faulty. We can check this property by changing the earlier network as follows:

- The first-stage data distributor is replaced by a faulty component (as we assume that the data source may have faults).
- One of the communications nodes is also assumed faulty (as we assume that one node shares an FCR with the data source).
- The final voter is replaced by a process which observes the values from the communications nodes, and signals an error if any cycle of communication does not include all working channels in agreement, but which does not actually distinguish which value is agreed upon. We may verify that the system satisfies the agreement property simply by showing that the error condition can never arise.

We can also exploit the symmetry of our system as described in the previous section, reducing the number of voters actually modeled to two: by showing that these are always in agreement we demonstrate agreement for any pair of fault-free outputs. The following variant of the model introduced in Section 3 incorporates these features:

ftagree.csp: Model demonstrating agreement of 4-FCR Oral-Messages algorithm in the presence of a single Byzantine fault.

(c) Formal Systems Design & Development, Inc, 1994 Originated by: Dave Jackson.
This version:
```
-- $Id: ftagree.csp,v 1.3 1995/04/25 00:17:16 dave Exp $
```

Once again, the basic data types are boolean:
```
RAWDATA = {0,1}
```
We also define an error value Err (which can be any value not in RAWDATA).
```
Err = 99
DATA = union({Err},RAWDATA)
```

Specification of data exchange mechanism

We define an event which indicates that a disagreement has been detected:

```
pragma channel Error
```

This will be the only output of our system, and our specification insists that even this should not occur, so our ultimate requirement is just that the system is equivalent to the process which performs only input actions:

```
SPEC = RUN({source.i | i <- RAWDATA })
```

Model of the OM algorithm for two voters of a 4-FCR system

The voters are identical to those of the previous model:

```
MAJ(sources, sink) = MAJORITY(sources, {}, {}, {}, sink)

MAJORITY(expected, zeroes, ones, errs, sink) =
  if (card(expected) == 0) then OUTPUT(zeroes, ones, errs, sink)
    else ([] x : expected @ (x?y ->
      (if (y==0)
        then MAJORITY(diff(expected,{x}), union(zeroes,{x}), ones, errs, sink)
        else if (y==1) then
             MAJORITY(diff(expected,{x}), zeroes, union(ones,{x}),errs,sink)
        else
             MAJORITY(diff(expected,{x}), zeroes, ones,union(errs,{x}),sink))))

OUTPUT(zeroes, ones, errs, sink) =
  if (card(ones) < card(zeroes))
```

```
    then sink!0 -> MAJ(Union({zeroes, ones, errs}), sink)
    else sink!1 -> MAJ(Union({zeroes, ones, errs}), sink)
```

Each of the 4 FCR's will still provide data to the voters, but we are now concerned with only two of the four outputs:

```
COPY = inp ? x:RAWDATA -> o0 ! x -> COPY

pragma channel inp : DATA
pragma channel o0, o1, o2, o3 : DATA

Insert =  (COPY [| {inp} |] (COPY[[ o0 <- o1 ]]))

Source =  COPY [| {inp } |]
         ((COPY[[ o0 <- o1 ]]) [| { inp } |]
         ((COPY[[ o0 <- o2 ]]) [| { inp } |]
         (COPY[[ o0 <- o3 ]])))
```

The overall structure of the model is unmodified:

Overall inputs and outputs

```
pragma channel source, sink : DATA
```

Peer-to-peer communications

```
pragma channel xcmid : DATA
pragma channel xc01, xc02, xc03 : DATA
pragma channel xc10, xc12, xc13 : DATA
pragma channel xc20, xc21, xc23 : DATA
pragma channel xc30, xc31, xc32 : DATA
```

The data source and sink of each communications node

```
pragma channel ain, bin, cin, din : DATA
pragma channel aout, bout, cout, dout : DATA
```

And the interface sets of each FCR:

```
XCI0 = {xc10, xc20, xc30}
XCI1 = {xc01, xc21, xc31}
```

```
XCI2 = {xc02, xc12, xc32}
XCI3 = {xc03, xc13, xc23}

XCO0 = {xc01, xc02, xc03}
XCO1 = {xc10, xc12, xc13}
XCO2 = {xc20, xc21, xc23}
XCO3 = {xc30, xc31, xc32}

ALPHAA = (Union({{ain, aout}, XCI0, XCO0}))
ALPHAB = (Union({{bin, bout}, XCI1, XCO1}))
ALPHAC = (Union({{cin, cout}, XCI2, XCO2}))
ALPHAD = (Union({{din, dout}, XCI3, XCO3}))
```

We now define processes representing the FCR's. According to their role in our model, we have three representations:

- *A functioning FCR whose output we study, with data distribution and voting components;*
- *A functioning FCR whose output is not analysed, containing just the data distribution element; and*
- *A faulty FCR whose data distribution element is unreliable (and whose voted output we do not model).*

Two fully-modelled FCR's

```
FtLaneA =
  ((Insert [[ inp <- ain, o0 <- xcmid, o1 <- xc01 ]])
  [(union({ain, xcmid}, XCO0))||(union({aout, xcmid}, XCI0))]
  (MAJ(union(XCI0,{xcmid}), aout)))
  \ {xcmid}

FtLaneB =
  ((Insert [[ inp <- bin, o0 <- xcmid, o1 <- xc10 ]])
  [(union({bin, xcmid}, XCO1))||(union({bout, xcmid}, XCI1))]
  (MAJ(union(XCI1,{xcmid}), bout)))
  \ {xcmid}
```

One partially-modelled non-faulty FCR:

```
FtLaneC =
  (Insert [[ inp <- cin, o0 <- xc20, o1 <- xc21]])
```

And a faulty FCR, which ignores all inputs and may produce arbitrary valid outputs, and which may also be observed to be faulty:

```
FtLaneD = RUN(Union({events(i)| i<- union({din},XCI3)}))
          ||| CHAOS(Union({events(i) | i <- union({|dout|},XCO3)}))
          ||| RUN({c.Err | c <- union({dout},XCO3)})
```

where RUN is the process which simply performs arbitary sequences of actions from the specified set:

```
RUN(A) = [] a:A @ a -> RUN(A)
```

The fault-tolerant communication system as a whole is a parallel combination of these, identical to the previous model:

```
FtBuff =
  (((FtLaneA [ALPHAA||ALPHAB] FtLaneB)
  [union(ALPHAA, ALPHAB)||union(ALPHAC,ALPHAD)]
    (FtLaneC [ALPHAC||ALPHAD] FtLaneD)) \ Union({XCI0, XCI1, XCI2, XCI3}))

ALPHAIN = {source, ain, bin, cin, din}
ALPHAFT = {ain, bin, cin, din, aout, bout, cout, dout}
ALPHAMJ = {Error, aout, bout, cout, dout}
```

Our observer process has a similar form to the majority voter, but can be simplified because we consider fewer inputs to it, and it need generate only the Error signal, when required.

```
Observer = aout ? x -> bout ? y -> Check(x,y) []
           bout ? x -> aout ? y -> Check(x,y)

Check(x,y) = if x == y then Observer else Error -> STOP
```

The system structure is identical to that of the earlier model, with the faulty data source and observer replacing the data distribution and voting elements:

```
System =
  (((Source [[inp <- source, o0 <- ain, o1 <- bin, o2 <- cin, o3 <- din]])
  [ALPHAIN||ALPHAFT]
  FtBuff)
  [(Union({ALPHAIN, ALPHAFT}))||ALPHAMJ]
  Observer) \ ALPHAFT
```

We may now verify that all behaviors System are behaviors of SPEC.

5 Fault-management & Application Programs

We have demonstrated in the preceding sections that tolerance to Byzantine faults can be achieved by designing our network of replicated processors to implement the Oral Messages algorithm. This tolerance is a property of the network and its communication pattern, and is thus independent of the actual application program, provided that sufficient data is exchanged and voted upon to keep the replicated copies of the program in agreement.

The second goal of our approach is to tolerate transient faults, including (but not limited to) those which affect a large proportion of our network for a brief interval. Designing and verifying strategies to achieve this aim will necessarily involve a more detailed knowledge of the operation of the application program than we have used in the earlier parts of this document. In particular, we will need knowledge of the tasks executed by the application program and their data dependency and scheduling constraints. We will adopt the view that real-time applications are typically constructed as a set of atomic tasks, exchanging data by means of shared variables, and subject to data-dependency and timing constraints as discussed in [9].

5.1 Permuted scheduling

One method identified in [1] to reduce the impact of multi-processor transient faults is to ensure that our replicated processors execute different tasks at each instant: rather than fix a schedule for executing application code, we define a set of *permuted schedules*. We intend that if a transient fault disrupts the tasks executing at a given time on a number of processors, then there should still be enough redundant executions of those tasks completed at other points in the same scheduler cycle for valid results to be obtained by voting.

5.1.1 Validity of Permutation

The potential benefits of permuted schedules can be verified using a relatively straightforward, if potentially unwieldy CSP model. The model given below characterize the

communications element of a system by a series of $VOTER$ processes, each concerned with validating the output of some task. They repeatedly obtain information (on a channel *task*) from the execution of a task; for simplicity we assume that it is clear from a boolean flag passed along this channel whether or not the execution succeeded[2]. Provided that at least two successful executions occur in each cycle, the voter will successfully agree on the output values of that task (signalled by the *pass* event) and wait for the end of a frame (indicated by the *sync* event).

The actual permuted schedules can be modeled in an abstract way by providing each task with a "source" of executions – we do not need to model the actual schedules explicitly, but only to capture the condition which a reasonable set of permutations will satisfy in the presence of transients: each task will be executed four times in each cycle, of which no more than one will be corrupted. This importance of this model is that in later documents we will be able to constrain these sources by placing them in parallel with particular schedulers, and verify that those schedulers do meet the following correctness condition: We can combine source processes for each of the tasks under consideration with the voters, and demonstrate that the voters are always satisfied that sufficient executions have succeeded. In terms of the model below we must show that each frame contains a *pass* event for all tasks.

timing.csp: A model supporting verification of permuted schedules and associated voting.

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Richard Chapman / Michael Goldsmith This version:
`-- $Id: timing.csp,v 2.0 1994/12/16 17:44:03 dave Del $`

Basic type definitions:

```
TASK = { 0, 1, 2, 3, 4 }          The set of task names
BOOL = { true, false }                  and validity values
```

Channel declarations:

The following channels indicate completion of a task instance, and pass a flag indicating its success or failure:
`pragma channel task : TASK . BOOL`

pass is used to indicate successful acquisition of sufficient correct copies of the output of a task by a voter

[2]This is simply an abstraction of the actual voting and comparison of data.

```
pragma channel pass : TASK
```

```
pragma channel sync
```
indicates the end of a frame, and

```
pragma channel work
```
is an interleaved event simply representing the occurence of some internal computation (typically related to comparing the results of different instances of a task).

The communication part of the system defines a voting process for each task in the system:

```
COMMS =                    ((((VOTER(0, 2)
      [| {| sync |} |] VOTER(1, 2))
      [| {| sync |} |] VOTER(2, 2))
      [| {| sync |} |] VOTER(3, 2))
      [| {| sync |} |] VOTER(4, 2))
```
The voters synchronize on the sync signal, ensuring that all tasks are validated with respect to the same cycle boundaries.

The voter process itself accepts inputs on task, and when sufficient valid instances have been counted, it outputs a pass signal recording the task number.

```
VOTER (i, n) =
        task.i ? ok ->
                if ok
                then        if n == 2
                        then VOTER (i, 1)
                        else if n == 1
                        then work -> pass ! i -> FRAME (i)
                        else work -> VOTER(i, n) never happens
else work -> VOTER (i, n)
```

After successful output, the voter waits for completion of the cycle. It is still prepared to accept (and discard) further completion signals.

```
FRAME (i) =
        (sync -> VOTER (i, 2)) [] task.i ? any -> work -> FRAME (i)
```

The tasks are represented by a combination of source processes, each enforcing the condition that sufficient correct instances of the appropraite task occur in each cycle. Their only interaction at present to to synchronize on the end-of-cycle signal.

Future models will exploit this framework by combining particular scheduler patterns in parallel with these processes.

```
SOURCES =              ((((SOURCE(0)
      [| {| sync |} |] SOURCE(1))
      [| {| sync |} |] SOURCE(2))
      [| {| sync |} |] SOURCE(3))
      [| {| sync |} |] SOURCE(4))
```

At the start of each cycle, four instances of the task are required, and no incorrect instances have been observed.

```
SOURCE (i) = NOTYETBROKEN (i, 4)
```

This process represents a source which has yet to observe an unsuccessful execution. It permits a synchronization signal and a return to its initial state if all four instances of the task have been observed. (n holds the number yet to be seen). If only one task remains to complete, it will allow the end-of-frame signal, assuming the last instance of the task to have failed. In other cases, it waits for a completion signal and decerments the counter if the execution succeed, or moves to the $ALREADYBROKEN$ state if it failed.

```
NOTYETBROKEN (i, n) =
        if n == 0
        then sync -> SOURCE (i)
        else if n == 1
        then sync -> SOURCE (i) [] task.i ? ok -> sync -> SOURCE (i)
        else task.i ? ok ->
                if ok
                then NOTYETBROKEN (i, n-1)
                else ALREADYBROKEN (i, n-1)
```

In each cycle, once a single erroneous execution has been observed, the remaining n must complete successfully.

```
ALREADYBROKEN (i, n) =
        if n == 0
```

```
then sync -> SOURCE (i)
else task.i ! true -> ALREADYBROKEN (i, n-1)
```

In the current version, our system model is simply the combination of the task execution part and the communication part.

```
SYSTEM = SOURCES [| {| sync, task |} |] COMMS
```

The model show above does have some practical disadvantages, however. The computation (represented by *work*), and the task completion (*task*) signals are arbitrarily interleaved, and the number of possible states whose behavior must be considered (either by automatic or manual analysis) grows very rapidly as the number of tasks considered increases. We can reduce this growth by noting that the voters would in practice differentiate between processing and communication, possibly refusing to exchange more data until the work associated with previous communication was complete. In the CSP model, we wish to distinguish the *work* events from the *task* communications by a difference in priority.

Encoding this distinction in a form suitable for use with the current **FDR** tool is quite difficult. We must consider the voters as constituting a single process which maintains a vector of information, holding a count of executions of each task in each element of the vector. This allows us to replace the interleaving of communications in the previous model with a sequential form which maintains the desired relationship between *task*, *work* and *pass* events. A model which uses the SML embedding techniques supported by **FDR** to implement this scheme is given in Appendix A.

A much more satisfactory model incorporating priorities to distinguish internal and external activity in a system or sub-system can be built on the basis of Dr Goldsmith's work described in another part of this report [5]. The **FDR** tool currently under development ([4]) will, when extended by the prioritization operator developed by this project and discussed in [5], allow such models to be written in the simpler style of the model given above, while maintaining the semantic distinctions and practical efficiency of that given in Appendix A. Such a framework will be essential for the extension of this framework into a tool for checking the transient-tolerance of a specified set of permutations of a schedule.

5.1.2 Permitted Permutations

For the use of permuted schedules to be valid, we must be able to find an appropriate number of viable schedules for the task set which makes up the application program. This potentially difficult task is subject to some non-obvious constraints, as we will show here. Consider the data dependence relation and four schedules in Figure 6. Assume all tasks take equal time to execute. The instance of task one in any given frame for processor four computes a value that will not be computed by the other

three processors until the beginning of the next frame. Suppose that we are voting on task seven only, and that processor four's value for task seven in some frame turns out to be in error. That means that the value already computed for task one for the next cycle must also be invalidated. Somehow, processor four must at some future point "catch up" – compute values for tasks one through six, since they are not voted, and be able to compute a valid value for task one in advance of the other three processors if it is to resume executing its schedule. Processor four will never be able to do it, since to do so it must after some number of frames k have computed $7k + 1$ values in $7k$ time slots.

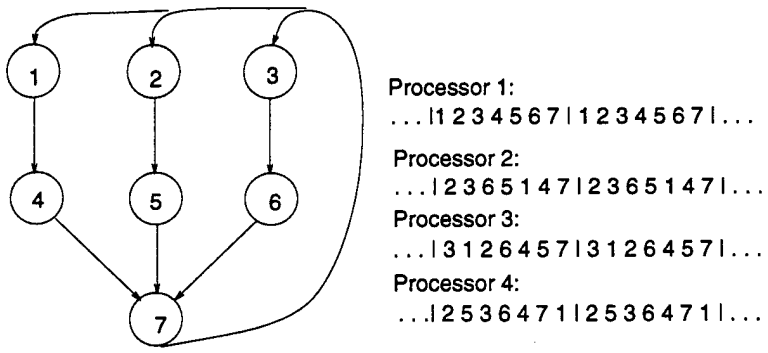


Figure 6: Disallowed set of permutations

Consequently we require permissible permuted schedules to obey the property that within a single frame the instances of a task on all replicated processors correspond to the same iteration. That is, if we let $O_k^p(t_i)$ represent the output from task t_i on processor p in frame k, then in the absence of failures,

$$\forall k \bullet \forall p' \in Processors \bullet O_k^{p'}(t_i) = O_k^p(t_i)$$

5.2 Voting in permuted schedules

The standard method of implementing systems whose state information is maintained by an interactive consistency algorithm such as Oral Messages is to arrange that applications use the agreed value of a state variable in place of the locally calculated one when a task requires that variable as input. In effect, we must arrange our voting and computation in such a way that a sufficient set of state values are always agreed by voting before they are used by tasks which depend on them.

These constraints further complicate the process of finding suitable schedules for a set of redundant processors which, as the previous section and reference [9] show, is already subject to significant constraints.

We therefore seek to relax this "vote before use" condition, in order to introduce sufficient flexibility to support permuted schedules and to gain other benefits:

- Relaxing the restriction places fewer constraints on the sequence in which tasks can be executed, potentially reducing any need for one processor to be idle while others compute values which need agreement.

- In systems where computation (by the application) and communication (which constitutes much of the voting process) use different resources, they can be overlapped to a greater extent if the strict ordering is relaxed, leading to significant performance benefits.

Rather than requiring the replicated processors to vote on the outputs of all tasks, we only specify voting on outputs to actuators and on a set of tasks satisfying a minimal voting condition ([11], p. 60), which we call a *basis set* of tasks. If permuted schedules were not permitted, voting could occur immediately upon completion of the task to be voted (by all the processors) which should happen simultaneously, given the requirement that we know absolute execution times for all tasks ([1], p. 1).

However, if permuted schedules are permitted, voting must be delayed at least until a plurality of processors have computed some output value for the task to be voted. The point within a frame at which a given basis task's output can be voted is statically determinable and thus the communication events necessary to carry out the voting can be incorporated into the schedule.

A consequence of the necessary delay in voting is that it becomes likely that a processor that is the first to run some basis task will have to use its locally computed, not yet voted, value for the output of that basis task until the voted value becomes available. If the voted value agrees with the locally computed value, all is well, but if the locally computed value is invalidated by the vote, the processor must begin recovery of a number of tasks. The results of not only the voted task but also all other comparable tasks (either as ancestors or descendants) in the transitive closure of the data dependence relation between tasks [7]become invalid, as in the example of Figure 7. Upon invalidation of any tasks, there must exist some sequence of actions that the recovered processor can take to restore all invalidated tasks to having valid input data at the appropriate times in each frame, according to its schedule.

We can shorten the waiting period required for voting by not requiring a task to wait for results from all four replicated processors. Two values in agreement are enough evidence for a processor to conclude that it has the voted value, so why wait for all four? However, a processor that proceeds before receiving input from all processors contributing to a vote must ensure that those messages it plans to ignore are properly dealt with if they should arrive at some future point. We propose that a processor deciding to ignore communications from some other processors should spawn a *sacrificial buffer* process that will catch those messages when they do arrive, or notify the processor if they never do (that is, if the buffer process receives another request from its own processor to wait on a value from the peer processor before it has received a first value from the peer). This fact is evidence of a failure either in

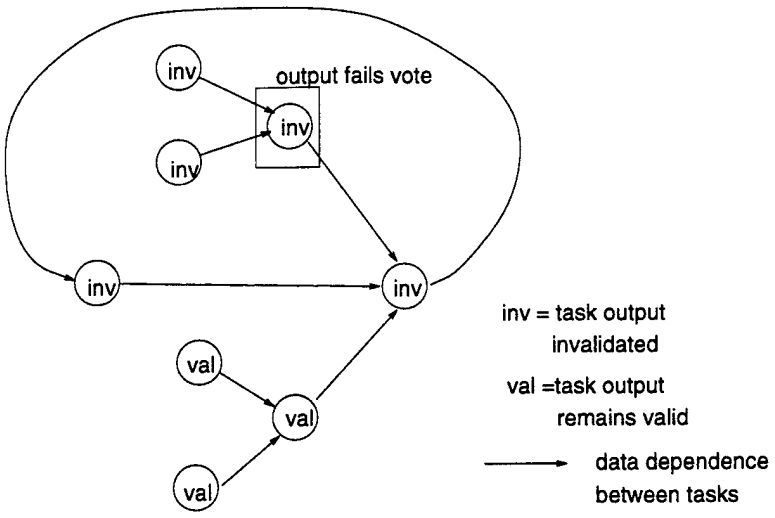


Figure 7: Tasks that must be invalidated

the peer processor or the link. We outline a specification for such a buffer and the code that the processor spawning the buffer must run in the next section of this note.

There are obviously a number of potential difficulties which relaxing the voting pattern in this way may introduce. It is obviously vital that the tolerance of Byzantine faults should not be reduced, and indeed this fact does follow from the properties of the network shown in Section 3. Because we do eventually obtain as much information on the correctness of a value as is available in the straight-forward implementation of the OM algorithm our ability to detect and correct errors is unaltered, although detection of an error may be slightly delayed when compared with a fully sequential voting arrangement.

The most significant penalty incurred by the change is that transient errors in the data held by a processor are no longer corrected "automatically": if state data is always agreed with a processor's peers before being used then a single corrupt value will not be passed to any instance of the tasks which use it, and if the fault causing corruption is transient it will be corrected when the value is next modified. This is obviously not the case if a processor continues using the corrupt value without checking it. The process of recovery from transient errors will be considered further in a later section.

6 Distributed Models of a Voter

Below we develop a model of a voting mechanism that can be used when processors are running permuted schedules. Rather than requiring the communications processor to spawn a process to catch "late" data values transmitted by peer processors, we

add four local processes, running concurrently with the voting mechanism, which we call *smart buffers.* Each smart buffer is responsible for reception of messages from one of the peer processors, for maintaining the local processors' decision about the state of that peer processor (good, faulty, or dead), and for conveying information about recent communications with the peer processor when requested.

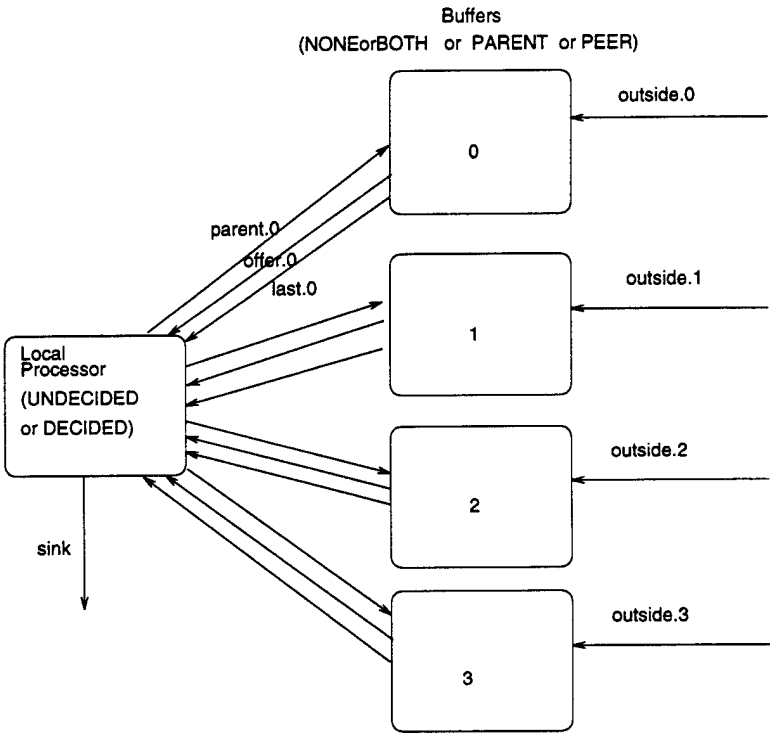


Figure 8: Local processor and buffers for communication with peers

The buffer has three major states. The current state is determined by which of the local (or parent) processor or the remote (or peer) processor it has last communicated with. The communications processor has two major states. We say it is *decided* if it has determined the valid value for its task for this frame as a result of comparisons between values sent by the remote processors for the task's value this frame. Otherwise it is *undecided* .

sb.csp: Distributed model of communication and voting

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Richard Chapman This version:

```
-- $Id: sb.csp,v 2.0 1994/12/16 17:44:03 dave Del $
```

There are four processors, numbered 0 to 3.

```
FOUR = {0, 1, 2, 3}
```

The value computed by a task may be one of two valid values, or a mesage from a processor to ignore its value

```
DATAVAL = { one, zero , ignoreme }
```

A processor may assign one of its peers any of the following status values

```
STATUSVAL = { bad, ok, dead }
```

The system present at each processor consists of the process running on the processor itself, plus four concurrent processes representing buffers to receive values from the remote processors (we could handle the value computed locally as a special case, but do not). The 4 buffer processes send data values to the local processor over the offer channels

```
pragma channel offer : FOUR . DATAVAL
```

The local processor can communicate its voted value for the task to the buffers over the parent channels

```
pragma channel parent : FOUR . DATAVAL
```

Channel last is used by a buffer to communicate the status of the peer with which it communicates to the local processor.

```
pragma channel last : FOUR . STATUSVAL
```

Sink is the channel on which the processor broadcasts a valid value for a task to the outside world, once per frame

```
pragma channel sink : DATAVAL
```

Synchronization signal sent between successive iterations of a task:

```
pragma channel frame
```

channels for peer processors to send values to sacrificial buffers:

```
pragma channel outside : FOUR . DATAVAL
```

Code for the local processor to interface with the sacrificial buffer

A processor determines the valid value for a task by receiving the values computed by its peer processors and comparing that value with its own locally-computed value for that task. Once a processor has received the same value from two processors (one of which could be itself), it can conclude that the value it received more than once is the valid value.

At any time, the communications processor will be running one of two processes (UNDECIDED or DECIDED) for each task.

The process UNDECIDED represents the state of the communications processor when it has not yet received two values in agreement for some task. The set I is the set of other processors from which the processor has yet to receive a value, and the set A is a set of ordered pairs (processor, value) that have been received. If a peer processor sends an ignoreme message, its number is removed from I. If a peer processor sends any other data value, that value is compared to previously received values. If that value is found in the list, the processor concludes that it has the valid value and behaves like process DECIDED (keeping track in its first parameter of which peer processes from which it has not yet heard), else it adds the value to the set of received values and behaves like UNDECIDED

```
UNDECIDED (I, A, untimed) =
        ([] i: I @ offer.i ? x ->

        if (x == ignoreme) then
                UNDECIDED (diff (I, {i}), A, untimed) else
        if (member (x, { head (tail (x1)) | x1 <- A })) then
                DECIDED (diff (I, {i}), x, untimed)
        else
                UNDECIDED(diff(I,{i}), union(A,{ <i,x> }), untimed)
        )
```

Once a process has decided the valid value for a task (parameter x in the process DECIDED, below) , it can use that value for further computation, but must rely on a process (a concurrently running "smart buffer") to handle reception of the remaining

transmissions of values for that task by the other peer processors (whose numbers are in set I). The smart buffer must also be responsible for notifying the local processor if any peer processors fail to respond

As soon as the local processor decides the valid value, it sends value over channel parent to the buffers, who will use it in determining the status (good, bad, or dead) of the peer processors

When the local processor has finished notifying the buffers, it announces the value it determined to be valid to the outside world over channel sink, then waits for the frame synchronization event and starts over.

```
DECIDED (I, x, untimed) =
        ([] i:I @ parent.i ! x -> last.i ? s ->
                 DECIDED (diff (I, {i}), x, untimed))
        []
        (if (empty (I)) then
                sink ! x ->
                        if untimed
                        then frame -> UNDECIDED (FOUR, {}, untimed)
                        else UNDECIDED (FOUR, {}, untimed)
        else STOP)
```

Code for a smart buffer running in parallel with a processor

The smart buffer has one of several states depending on whom it heard from last: the PARENT (local) processor, the PEER (remote or local) processor, or NONEorBOTH (ready to receive a message from either).

Initially, a smart buffer has not heard either from its parent (via channel parent) or from any peer processor (via channel outside). It is ready to communicate via either channel, and change its state based on which it hears from first

```
NONEorBOTH (i, s, untimed) =
        (parent.i ? x -> last.i ! s -> PARENT (i, x, untimed))
        []
        (outside.i ? y -> PEER (i, y, s, untimed))
```

A smart buffer that has last heard from its parent knows the value the parent decided was valid (x), and is waiting to hear that value from the PEER processor. If it does

hear a value from the peer, it computes a statusval for the peer (okay or bad, depending if the value sent by the peer is the same as that decided upon by the parent) and resumes listening for either the parent or the peer

If the frame event occurs before the buffer hears from the peer, it assumes the peer is dead and changes its status accordingly. If frame events are not being used, if the buffer hears from the parent again before hearing from the peer, it sends a message to the parent (over channel last) indicating that it believes the peer is dead

```
PARENT (i, x, untimed) =
        (outside.i ? y ->
                if untimed
                then frame ->
                  NONEorBOTH (i, if x==y then ok else bad, untimed)
                else
                  NONEorBOTH (i, if x==y then ok else bad, untimed))
        []
        if untimed
        then frame -> NONEorBOTH (i, dead, untimed)
        else parent.i ? xx -> last.i ! dead -> PARENT(i, xx, untimed)
```

A buffer that has heard from the peer processor sends the value it heard to the parent via the offer channel. After sending an offer it waits for the frame synchronization event and then resumes waiting to hear from either the parent or the peer

If the processor receives a value from the parent before it can offer the value from the peer to the parent, obviously the parent already had enough values from other buffers to make a decision, so the buffer sends the status value from the last frame to the processor and then computes a new status value for this frame, arrived at by comparing the value received from the parent this frame with the value received from the peer this frame, then waits for the frame synchronization event, and then listens for either the parent or peer at the start of the next frame

```
PEER (i, y, s, untimed) =
        (offer.i ! y ->
                if untimed
                then frame -> NONEorBOTH (i, ok, untimed)
                else NONEorBOTH (i, ok, untimed))
        []
        (parent.i ? x -> last.i ! s ->
                if untimed
```

```
                then frame ->
                  NONEorBOTH (i, if x==y then ok else bad, untimed)
                else
                  NONEorBOTH (i, if x==y then ok else bad, untimed))
```

The system consists of a processor (initially running UNDECIDED(FOUR,,true) and four smart buffers, one for each of the four peer processors. We could optimize this to three and handle the locally computed value entirely within the local processor if desired

```
uSYSTEM = UNDECIDED (FOUR, {}, true)
          [| {| last, offer, parent, frame |} |]
          ((((                    NONEorBOTH (0, ok, true))
            [| {frame} |] NONEorBOTH (1, ok, true))
            [| {frame} |] NONEorBOTH (2, ok, true))
            [| {frame} |] NONBEorBOTH (3, ok, true))
```

When we hide the communication between the four buffers and the local processor we get:

```
UntimedSystem = uSYSTEM \ {| last, offer, parent |}
```

If we dispense with the frame synchronization events:

```
tSYSTEM = UNDECIDED (FOUR, {}, false)
          [| {| last, offer, parent |} |]
          ((    NONEorBOTH(0,ok,false) ||| NONEorBOTH(1,ok,false)
            ||| NONEorBOTH(2,ok,false) ||| NONEorBOTH(3,ok,false))
           [| {| frame |} |] frame -> ZERO)

SystemWithoutTiming = tSYSTEM \ {| last, offer, parent |}
```

In order to assert that frame never occurs, the process above includes a transition to ZERO if frame should ever occur. Because ZERO is the "worst-possible" process in the Failures-Divergence model, this will result in tSYSTEM failing any non-trivial refinement check, should frame occur.

```
ZERO = ZERO |~| ZERO
```

The bottom process is represented as a non-deterministic choice for purely technical

*reasons. (***FDR***cannot itself successfully compile the more usual definition*
`-- ZERO = ZERO`*).*

The specification for the System described above

MAJORITY's three parameters are sets of processes. I represents the processors which have not contributed a value for the task this frame, while Zeroes and Ones are respectively, the sets of processors that have contributed a value of zero and a value of one this frame (note that a processor sending an ignoreme message drops out of I without ever appearing in Ones or Zeroes that frame)

If the size of Zeroes exceeds one, the specification may output a value of one on channel sink. It may output a zero on sink if the size of Zeroes exceeds one. If both sets exceed one in size, the specification produces a nondeterministic result

If the set I is not empty at the time the specification sends its decision on the valid value on channel sink, the specification must behave like process CHOMP until the end of the frame

CHOMP acts as a buffer to receive any values transmitted by the peer processors after the local processor has produced output during that frame on channel sink (in the system, the smart buffers handle this)

```
MAJORITY (I, Zeroes, Ones) =
        (outside ? i:I ? x ->
             \    if (x == zero)
        then MAJORITY (diff (I, {i}), union (Zeroes, {i}), Ones)
                else if (x == one)
                then MAJORITY(diff(I, {i}), Zeroes, union(Ones, {i}))
                else MAJORITY(diff(I, {i}), Zeroes, Ones))
        []
        ((if 1 < card (Ones)
          then sink ! one -> CHOMP (I)
          else STOP)
         |~|
          if 1 < card (Zeroes)
          then sink ! zero -> CHOMP (I)
          else STOP)
        []
        if empty (I)
        then (if card (Ones) < card (Zeroes)
```

```
            then sink ! zero -> CHOMP (I)
            else if card (Zeroes) < card (Ones)
            then sink ! one -> CHOMP (I)
            else (sink ! zero -> CHOMP (I) |~|
                  sink ! one -> CHOMP (I)))
        else STOP

CHOMP(I) =
        (outside ? i:I ? x -> CHOMP (diff (I, {i})))
        []
        frame -> MAJORITY (FOUR, {}, {})
```

Initially, no processors have sent values of one or zero, and the specification is waiting on a value from all four processors

```
SPEC = MAJORITY (FOUR, {}, {})
```

Here we model the data distribution phase. A value on channel source is copied to each of the four outside channels by processes INJECTOR and SPREAD. A frame synchronization event is expected between sucessive inputs on channel source (and consequently between any successive pair of outside.i events).

```
pragma channel source : DATAVAL

 INJECTOR = source ? x -> SPREAD (FOUR, x)

SPREAD (I, x) =
        if empty (I)
        then frame -> INJECTOR
        else |~| i:I @ outside.i ! x -> SPREAD (diff(I, {i}), x)

FRAMESYNCH =
        (INJECTOR [| {| outside, frame |} |] UntimedSystem) \
        {| outside, frame |}
```

We introduce the possibility of a fault on channel outside.0 The same symmetry arguments previously made apply here.

```
XFRAMESYNCH =
        ((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
        [| {| outside, frame |} |]
```

```
        UntimedSystem) \ {| outside, frame |}

CYCLICINPUT =
        (INJECTOR [| {| outside |} |] SystemWithoutTiming) \
        {| outside, frame |}

XCYCLICINPUT =
        ((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside, frame |}

pragma channel forward : FOUR . DATAVAL
```

A 1-place buffer is introduced along each of the four outside.i channels, fed by the INJECTOR

```
BOUNDEDDELAY1 =
        ((INJECTOR[[outside<-forward]]
         [| {| forward |} |]
         (   BBUFFc(1,forward.0,outside.0)
         ||| BBUFFc(1,forward.1,outside.1)
         ||| BBUFFc(1,forward.2,outside.2)
         ||| BBUFFc(1,forward.3,outside.3))
        ) \ {| forward, frame |}
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside |}
```

The possibility of a fault is allowed on channel outside.0

```
XBOUNDEDDELAY1 =
        (((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
         [| {| forward |} |]
         (   BBUFFc(1,forward.1,outside.1)
         ||| BBUFFc(1,forward.2,outside.2)
         ||| BBUFFc(1,forward.3,outside.3))
        ) \ {| forward, frame |}
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside |}

pragma channel tock
```

REGULATOR runs in parallel with the distribution of values to the outside channels to guarantee that all transmissions of values on the the outside.i channels (i is an element of parameter I) occur within N tocks, and that following the completion of one frame of events on the outside channels, M tocks elapse before the outside events for the next frame commence

```
REGULATOR (I, M, N) =
        tock -> REGULATOR (I, M, N) []
        outside ? i:I ? x -> REGULATOR' (I, M, N, diff (I, {i}), 0)
```

J is the subset of I whose outside channels have had no event this frame yet, while n is the number of tocks elapsed since beginning of this frame's outside events

```
REGULATOR' (I, M, N, J, n) =
        (if n < N
         then tock -> REGULATOR' (I, M, N, J, n+1)
         else STOP)
        []
        (outside ? i:I ? x -> REGULATOR' (I, M, N, diff (J, {i}), n))
        []
        (if empty (J)
         then DELAY (M); REGULATOR (I, M, N)
         else outside ? i:I ? x -> REGULATOR'(I,M,N,diff(J,{i}),n))
```

DELAY ensures that n tocks elapse between last outside event of one frame and first outside event of the next frame

```
DELAY (n) = if 0 < n then tock -> DELAY (n-1) else SKIP
```

Adding the REGULATOR to the rest of the already-developed system gives:

```
REGULATED1 =
        (((INJECTOR[[outside<-forward]]
          [| {| forward |} |]
          (   BBUFFc(1,forward.0,outside.0)
          ||| BBUFFc(1,forward.1,outside.1)
          ||| BBUFFc(1,forward.2,outside.2)
          ||| BBUFFc(1,forward.3,outside.3))
          ) \ {| forward, frame |}
         [| {| outside |} |]
         REGULATOR (FOUR, 2, 2)) \ {tock}
```

```
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside |}
```

and if we allow channel outside.0 to be faulty:

```
XBOUNDEDDELAY1 =
        (((INJECTOR \ {| outside.0 |} ||| CHAOS ({| outside.0 |}))
         [| {| forward |} |]
         (   BBUFFc(1,forward.1,outside.1)
         ||| BBUFFc(1,forward.2,outside.2)
         ||| BBUFFc(1,forward.3,outside.3))
        ) \ {| forward, frame |}
        [| {| outside |} |]
        SystemWithoutTiming) \ {| outside |}
```

Suppose we want to add the restriction that an event occurs on each of the four outside channels every k tocks, where k can vary from frame to frame, but must always be within some bounds of N tocks. Say, for some other integers A and B, that k can never be less than N - A nor more than N + B . Process PWB guarantees that rate of events on channel c:

```
PWB (c, N, B, A) = PWB' (c, N, B, A, 0)

PWB' (c, N, B, A, n) =
        if n < N - B
        then tock -> PWB' (c, N, B, A, n+1)
        else if n < N + A
        then     (tock -> PWB' (c, N, B, A, n+1))
             |~| (c ? x -> PWB' (c, N, B, A, n-N))
        else c ? x -> PWB' (c, N, B, A, n-N)

PWBs =            (((PWB (outside.0, 5, 2, 2)
        [| {tock} |] PWB (outside.1, 5, 2, 2))
        [| {tock} |] PWB (outside.2, 5, 2, 2))
        [| {tock} |] PWB (outside.3, 5, 2, 2))
```

Adding the restrictions on the buffers to the system yields:

```
PWB1 =  ((INJECTOR[[outside<-forward]]
          [| {| forward |} |]
          (   BBUFFc (1,forward.0,outside.0)
```

```
          ||| BBUFFc (1,forward.1,outside.1)
          ||| BBUFFc (1,forward.2,outside.2)
          ||| BBUFFc (1,forward.3,outside.3))
         ) \ {| forward, frame |}
        [| {| outside |} |]
        (PWBs \ {tock}
         [| {| outside |} |]
         SystemWithoutTiming)) \ {| outside |}

RUN(X) = [] a:X @ a -> RUN(X)
```

But we must also model the possibility that our faulty channel (channel outside.0) does not produce its values in a timely fashion:

```
XPWBs =            ((PWB (outside.1, 5, 2, 2)
       [| {tock} |] PWB (outside.2, 5, 2, 2))
       [| {tock} |] PWB (outside.3, 5, 2, 2))
```

The system becomes:

```
XPWB1 = ((INJECTOR[[outside<-forward]]
          [| {| forward |} |]
          (   RUN ({|forward.0|}) ||| CHAOS ({|outside.0|})
          ||| BBUFFc (1,forward.1,outside.1)
          ||| BBUFFc (1,forward.2,outside.2)
          ||| BBUFFc (1,forward.3,outside.3))
         ) \ {| forward, frame |}
        [| {| outside |} |]
        (XPWBs \ {tock}
         [| {| outside |} |]
         SystemWithoutTiming)) \ {| outside |}
```

Generic 1-place buffers

An N-place buffer receiving input on channel source and producing output on channel sink:

```
BBUFF (N) = BBUFFc (N, source, sink)
```

BBUFFc is an N-place buffer also taking the names of its input and output channels as parameters:

```
BBUFFc (N, in, out) = BBUFF' (N, 0, <>, in, out)
```

There are two more components to the state of BBUFF': it's current contents (the list of values s) and the number of items currently stored in the buffer:

```
BBUFF' (N, n, s, in, out) =
        if n == 0
        then in ? x -> BBUFF' (N, 1, <x>, in, out)
        else if n == N
        then out ! head (s) -> BBUFF' (N, n-1, tail (s), in, out)
        else
          ((out ! head (s) -> BBUFF' (N, n-1, tail (s), in, out))
        [] ( in ? x -> BBUFF' (N, n+1, s^<x>, in, out)
            |~| out ! head (s) -> BBUFF'(N,n-1,tail(s),in,out)))
```

7 Recovery from transient errors

In order to arrange that a process can recover from a transient error even if values are not always agreed before use, we must arrange that:

- An FCR which has suffered a transient fault can detect the resulting error and thus take appropriate recovery action.
- While such a node is recovering the erroneous values, it does not promote failure in the other FCRs in the system.
- During recovery, all significant state values will be recovered from correctly functioning peer nodes, and sufficient computation will be performed to maintain and re-generate state which is not directly communicated.

The first two requirements are relatively undemanding; the first is effectively a constraint on the types of errors that we can expect to tolerate. One implication which must be considered, however, is that the design will have to distinguish between a "local" value and a value received from a peer when performing comparisons[3]. If the "local" value disagrees with the majority, then a node should be considered to have suffered a fault and should attempt to recover the relevant values. The second requirement is also trivial for some classes of faults. If no further errors occur in the

[3]Note that the models in Section 3 did not need to make this distinction.

portion of the system state related to that which we are attempting to recover (using the dependency relationship discussed in Section 5.2), we are guaranteed that three correct values will always be available to non-faulty nodes, and thus any erroneous output from a processor in the process of recovering will be ignored. We can gain some advantage, however, from allowing a processor which has detected a transient fault to notify its peers of this fact: the three remaining FCRs may then be able to survive a second non-independent error by moving to a 2-out-of-3 voting scheme.

The last requirement in the above list is the most difficult to satisfy. If, for reasons of timing drift or because of a permuted schedule, a processor suffering a transient fault was the earliest node to compute the relevant values in each frame, we may never be able to guarantee that it can obtain a timely, reliable value from its peers. We can suggest several approaches to this problem, including

- Arranging that the entire system reverts to a fall-back schedule which does guarantee to agree all state values by voting.
- Finding (where possible) a "reversionary" schedule for the failed processor alone.
- Arranging that the recovering processor and one of the fault-free peers change to pair of reversionary schedules which transfer corrected data to the recovering node while maintaining just sufficient of the normal behavior to ensure correct system operation.

These possibilities are discussed below:

7.0.1 A fully-voted reversionary schedule

We might propose the following scheme of operation: whenever it is determined that a processor has computed an invalid result for some task, that processor will be asked to sit idle until the end of the current frame, at which time all processors will stop running their particular permuted schedules and start running a single already-agreed-upon schedule. Thus, during the next frame, all processors will be running the same schedule. We know because of our requirement from Section 5.1.2 that only permitted permutations were used, that the output records for all non-invalidated replicated instances of all tasks contain the same data, and that if the invalid processor has not failed, it can recover valid values for all tasks (values derived from voted values of all tasks in the basis set) by the end of the next frame. If no notifications of invalid results are received by any processors during the execution of the recovery frame, the processors switch to their particular permuted schedules at the end of that recovery frame.

This scheme does, however, limit many of the advantages which motivate our use a relaxed voting scheme. If we must be able to operate on a fully-voted schedule, we cannot take advantage of the performance benefits which overlapping communication

and computation will bring. In particular, there will be many sets of permuted schedules for which no suitable fixed schedule with complete voting will exist. Further, at least for the duration of the recovery, we will have lost the benefits of permuted scheduling.

7.0.2 A single reversionary schedule

This alternative again suffers from the disadvantage that for many applications there will be no single order of task execution which requires data only after it has been made available by one of the peer nodes. If only the recovering processor reverts to this schedule, however, we can remove some of the constraints limiting our execution order. In particular, the reversionary schedule need not calculate any outputs or other values which are calculated afresh in each cycle; it need only perform that minimum computation which is necessary to maintain the relevant node state. In terms of the data dependency graph, we need only execute those tasks which occur on the cycles through the initial erroneous task. Branches which do not form part of a cycle may be neglected, and indeed as we saw above, there are concrete advantages to be gained from a processor informing its peers that it should be ignored in any votes which take place during its recovery.

Due to the difficulty of finding a suitable schedule (if one exists), this technique will obviously be limited in its application, particularly as to exploit the potential benefits of permuted scheduling, we must find a number of recovery schedules, each capable of re-generating a particular set of corrupt values while maintaining the outputs and correct behavior of the uncorrupted elements of the application.

7.0.3 Partial reversionary schedules

In an attempt to avoid some of the difficulties associated with both of the above schemes, we propose considering a method which combines some features from each. Finding a single processor schedule which is compatible with the permuted schedules already running on fault-free processors, as required by the previous scheme is clearly more difficult than the problem of finding two schedules which suffice to transfer some part of the system schedule to the recovering processor. This latter problem is simplified further if we allow the fault-free partner in such a recovery to neglect some of its output calculations (on the basis that there will still be duplicate correct values generated by the remaining pair) – we clearly lose tolerance to further faults during this operation, but the practical value of such resilience to two faults will obviously depend on the reliability analysis of a particular application. We do require a mechanism for identifying which processor should assist in the recovery when a fault is detected, but even here we may have a degree of choice over which of the remaining permuted schedules is most suitable for correcting the specific error identified.

The greatest cost of this approach is the effort of identifying sufficiently many reversionary schedules to maintain the benefits of temporal redundancy. We should note, however, that this is a task which is determined entirely by the static schedules chosen, and thus need not be carried out in a time-critical environment. The run-time penalty should be little more than identifying which regions of the data-dependency graph have been invalidated and looking up the appropriate recovery schedules in a pre-computed table.

To ease this task, it is perhaps desirable to consider the data-dependency graph as being divided into *software containment regions* which are treated as either being believed correct or believed corrupted as a whole. These regions must obviously contain the transitive closure of the relevant voted state variables, as discussed in Section 5.2. We also note that the fault-free processors initiating a recovery must agree on the identity of the FCR to be recovered and on the particular peer who will enter the assisting reversionary schedule. This information is, however, amenable to voting in a similar manner to other values, and is only required when votes are taken on state data – it need not apply to the agreement of output values, for example.

8 Conclusions

We do not expect this document to be viewed as a complete analysis of the FTP design, but to be seen as a working paper describing the state of various threads of analysis and modeling. One of the primary purposes of this paper is indeed, to present some ideas for comment from Draper representatives who, we hope, will be able to view them in the context of their greater familiarity with the concerns of the application domain. Significant features of recent developments include:

- Clarification of arguments based on symmetry which can be used to establish properties of the full FTP system from properties of a single voter. This work is sufficiently established that we feel a formal mathematical proof of the approach could be given. It is a result which will be particularly important in the future development of models which include more detail about the operating mechanisms of their components. It has already assisted in the rest of this work.

- Moving toward a less abstract model bearing a closer resemblance to the implementation, we have gained significant understanding of the problems faced in several key areas. These include

    - tolerance of transient faults,
    - recovery after transient errors, and
    - the benefits to be gained from temporal redundancy and permuted scheduling.

- We have presented models of the FTP consistency algorithm which include explicit timing information in both synchronous and semi-asynchronous models. These models include sufficient information about the communication mechanism to investigate the need for non-blocking and sacrificial buffers. We feel that these models approach the "Synchronous replicated" and "Asynchronous distributed" views of [1], although they still involve significant abstraction from the way in which the processing and voting elements operate, and the issue of establishing co-ordinated global timing has still to be addressed in detail.

The modeling which we have completed in this area is still highly abstract, but it provides important framework elements, and highlights those areas which place additional emphasis on new theories and tools.

The major prospects for future work on the demonstrator application lie in the following areas

- Our models are still very abstract in some areas: our models of communication are relatively close to transputer style implementations, but areas such as timing, clock synchronization and the mechanisms connecting hardware and software could benefit from more detail. Additional information may well allow us to relax some of our design constraints: for example our asynchronous timing model requires large margins in the specification of time-outs and cycle lengths, whereas slight improvements to the design we are formalizing may allow these margins to be reduced.

- At the implementation level, more detailed models of the interaction between software tasks is required, both in terms of specifying application timing constraints and especially in the relationship between communications hardware and software.

- The interface between communication, voting, and application software schedules is perhaps in greatest need of further formalization. Both this area and more general timing and scheduling issues will require the ability to model and distinguish systems using multi-processing on a single CPU and communications hardware supporting a single processor, as well as the theoretically simpler case of true multi-processor systems.

These prospects highlight some points of importance in the tool-development part of the project, in particular in the area of prioritization (as noted in Section 5.1.1) and possibly in assisting the modeling the interaction between varied hardware and software environments.

References

[1] N.A. Brock. Real-Time Scheduler: Natural Language Problem Statement. Technical report, Charles Stark Draper Laboratory, Inc., 1994. Deliverable D2.1 of SBIR N00014-93-C-0213, in [6].

[2] Neil A. Brock and Sharon L. Donald. Discussion of Errors and Their Effects on a HRT Scheduler. Technical report, The Charles Stark Draper Laboratory, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [7].

[3] Formal Systems (Europe) Ltd, 3 Alfred St, Oxford OX1 4EH, UK. *Failures-Divergences Refinement* (**FDR**), *User Manual and Tutorial*, 1994. Contact D.M. Jackson; Tel: [+44] (0)1865 728460, Fax [+44] (0)1865 201114, E-mail: `dave@fsel.com`.

[4] P.H.B. Gardiner and M.H. Goldsmith. Inside **FDR 2**. Technical report, Formal Systems Design & Development, Inc., 1994. Adjunct to D1.2 of SBIR N00014-93-C-0213, in [8].

[5] M.H. Goldsmith. A CSP Priority Operator for **FDR 2**; Prototype Software for Discrete Real-time Extensions to **FDR**. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [8].

[6] M.H. Goldsmith et al. *N00014-93-C-0213: Second Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[7] M.H. Goldsmith et al. *N00014-93-C-0213: Third Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[8] M.H. Goldsmith et al. *N00014-93-C-0213: Fourth Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[9] David M. Jackson and M.H. Goldsmith. Specifying Task Management; Single Processor Systems. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverables D 2.2 and D 2.3 of SBIR N00014-93-C-0213, in [7].

[10] Patrick Lincoln and John Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Proceedings of 23rd Fault-Tolerant Computing Symposium*, 1993.

[11] B.L. Di Vito, R.W. Butler, and J.L. Caldwell. Formal Design and Verification of a Reliable Computing Platform For Real-Time Control – Phase 1 Results. Technical Memorandum 102716, NASA, 1990.

A Vector-based Model for Permuted Scheduling

abstract.csp: An FDR-1 model of a voter for permuted schedules

(c) Formal Systems Design & Development, Inc, 1994

Originated by: Michael Goldsmith This version:
`-- $Id: abstract.csp,v 2.0 1994/12/16 17:44:03 dave Del $`

Define two vector operators in Standard ML: getnth returns the selected component of a sequence, setnth sets the selected component of the sequence to be the value specified.

Declare the function names as non-CSP definitions

```
pragma opaque "ML" getnth
pragma opaque "ML" setnth
```

Include the ML source code for the function implementations

```
pragma inline "ML" local
pragma inline "ML"   fun MLgetnth (0, a::x) = a
pragma inline "ML"     | MLgetnth (n, _::x) = MLgetnth (n-1, x)
pragma inline "ML"     | MLgetnth _ = raise SemanticError
pragma inline "ML"                    ("getnth: index too large")
pragma inline "ML" in
pragma inline "ML"   fun CSPgetnth [n, s] =
pragma inline "ML"     let val MLs = CheckSeq s
pragma inline "ML"         val MLn = NumberOf (CheckAtom n)
pragma inline "ML"     in MLgetnth (MLn, MLs)
pragma inline "ML"     end
pragma inline "ML"     | CSPgetnth x = raise TypeError
pragma inline "ML"        ("getnth: expected <number,sequence>,"
pragma inline "ML"        ^ " found "
pragma inline "ML"        ^ print_expression (EXPseqcomp (x, [])))
pragma inline "ML" end;
```

The following definition includes a call to print merely as development aid.

```
pragma inline "ML" local
pragma inline "ML"   fun revonto (a::x, y) = revonto (x, a::y)
pragma inline "ML"       | revonto (_, y)    =
pragma inline "ML"          (print "\nSTATE ";
pragma inline "ML"            map(print o print_expression)y;
pragma inline "ML"            print "\n"; y)
pragma inline "ML"   fun MLsetnth (0, _::x, v, y) = revonto (y, v::x)
pragma inline "ML"     | MLsetnth (n, a::x, v, y) =
pragma inline "ML"          MLsetnth (n-1, x, v, a::y)
pragma inline "ML"     | MLsetnth _ = raise SemanticError
pragma inline "ML"                    ("setnth: index too large")
pragma inline "ML" in
pragma inline "ML"   fun CSPsetnth [n, s, v] =
pragma inline "ML"     let val MLs = CheckSeq s
pragma inline "ML"         val MLn = NumberOf (CheckAtom n)
pragma inline "ML"         val _ = NumberOf (CheckAtom v)
pragma inline "ML"     in EXPseqcomp (MLsetnth (MLn, MLs, v, []), [])
pragma inline "ML"     end
pragma inline "ML"     | CSPsetnth x = raise TypeError
pragma inline "ML"       ("setnth: expected number,sequence,number>,"
pragma inline "ML"       ^ " found "
pragma inline "ML"       ^ print_expression (EXPseqcomp (x, [])));
pragma inline "ML" end;
```

Declare the relationship between the CSP names and the ML functions

```
pragma inline "ML" DefineMLFunction "getnth" CSPgetnth;
pragma inline "ML" DefineMLFunction "setnth" CSPsetnth;
```

The following sets and channels are equivalent to those in timing.csp

```
TASKS = { 0, 1, 2, 3, 4 }
BOOL = { true, false }
pragma channel task : TASKS . BOOL
pragma channel pass : TASKS
pragma channel work, sync
```

The communication model is now a single process with a vector argument

```
COMMS = JUDGE (<2,2,2,2,2>)
```

Initially this process accepts a termination signal, and if it is valid, moves to the DSZ state to decrement the appropriate value. If the execution was invalid, it performs the associated work (actually an abstraction of the decision process), and remains ready to accept another termination. Note that inputs are not accepted while work is being offered: this captures the prioritization of the "internal" action work over external communication

```
JUDGE (s) =
        task ? i ? b ->
                if b
                then DSZ (s, i, getnth (i, s))
                else work -> JUDGE (s)
```

This process examines the count relating to task i and performs appropraite action. If the recent termination we the first successful one, the counter is decremented (without doing any work for the comparison), and the JUDGE returns to its initial state. If one previous successful execution had preceded this one, a comparison is performed, and the successful acquisition of good data is signalled on pass. Further successful executions are ignored (after the comparison, which is necessary to detect the occurence of a transient error, although not to determine the actual value required).

```
DSZ (s, i, si) =
        if si == 2
        then JUDGE (setnth (i, s, 1))
        else if si == 1
        then work -> pass ! i -> FRAME (setnth (i, s, 0))
        else work -> JUDGE (s)
```

When a pass signal has been indicated, we examine the new values of all the counters in s to see if they are now all zero. (This uses the FDR set operator.) If this is the case, further tasks are ignored after a comparison, and the end-of-frame synchronization may occur. Otherwise the sub-system returns to its initial state.

```
FRAME (s) =
        if set (s) == { 0 }
        then (sync -> COMMS) [] task ? any -> work -> FRAME (s)
        else JUDGE (s)
```

This is Release 3.0 of this document, last modified by Michael Goldsmith at 20:35:05 GMT on April 25, 1995.

FormalSystems

Verifying Timing Properties of Static Schedulers

David Jackson
Formal Systems

April 25, 1995

Summary

This document describes the process of modelling a general class of real-time programs with cyclic, non-preemptive schedulers. We show that a large number of the requirements placed on these systems in real embedded-system applications can be captured as instances of a few general formal specifications. We also describe models of these programs in CSP which have the property that relatively few parameters need to be altered to reflect a change in schedule, most of the model being determined by the software architecture adopted.

Such models and requirements are ideally suited to mechanical verification, which can be carried out relatively inexpensively. We suggest a modification to the style of earlier deadline specifications in CSP which allows the "degree" of failure to be judged when a violation of the specification is identified in this way. In an extension to other work on this topic, we describe how the mechanical verification process can yield quantitative information about timing margins and processor utilization in addition to the qualitative verification of satisfaction.

We give an example based on a simplified, but realistic, automotive engine management system.

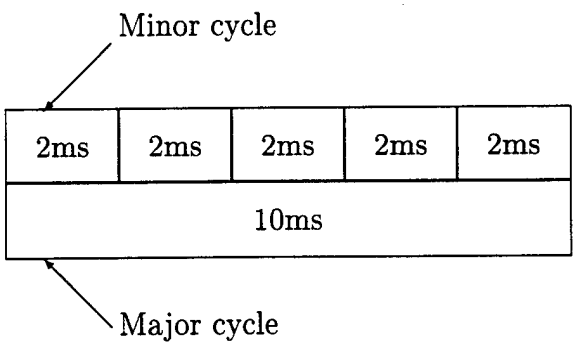
1 Background

A common feature of a number of embedded systems of which Formal Systems has experience is a scheduling scheme based on a statically determined cycle of task executions. These schedulers, and a number of more flexible schemes based on relatively

This is a working paper (W1.4.1) in the Office of Naval Research SBIR Project N00014–93–C–0213 *Embedded Transputer-based System Design.*

slow or constrained dynamic variation of a task cycle, are particularly amenable to formal and mechanical analysis. This document describes the specification of such systems in CSP, and their verification using the FDR tool.

As discussed in a number of documents relating to the Transputer Fault-Tolerant Processor ([1]), the time-critical tasks of an embedded system are commonly organized into a series of frames (of fixed duration), each of which is executed in a longer cycle. For example, if a control system requires that certain outputs are updated every 2ms, while other status information need only be computed every 10ms, we might structure the execution schedule as follows: The critical performance criteria



of embedded systems are principally defined by the points are which external inputs are monitored and external outputs are provided; the importance of such externally observed behavior makes such systems obvious candidates for methods (such as the CSP formalism) which treat the relationships between external events as paramount. This document will examine the type of constraint which might appear in the requirements specification of such a system, the structures which a real-time program might use to address them, and the verification of these properties using the **FDR** tool.

2 A Motivating Example

As an example of the type of system which we plan to analyze, consider an automotive Engine Management System (EMS). We will assume that this micro-processor system is responsible for monitoring a range of inputs from the engine, vehicle, and driver, including

- Crankshaft position (° from TDC)
- Crankshaft speed (averaged over some interval)
- Desired engine speed (from driver or transmission control)
- Fuel injector pressure (to check correct function)

- Engine cooling system temperature

and perhaps also

- Engine oil temperature
- Exhaust composition (O_2 %)
- etc.

It will produce outputs including

- Injector actuation signals
- Ignition timing signals (advance/retard)
- Fuel pump control signal
- Cooling fan start/stop signal
- Tachometer reading (for driver)
- Warning indications

We expect that some signals (Injector control) must be generated at a relatively high frequency (1 KHz) and that the majority of the others are required less frequently. Note that we are more concerned with providing examples of the features of a range of systems than with providing a realistic model of a particular application.

3 Requirements

Requirements are placed on an embedded system by the external engineering disciplines governing the application. A system will typically be required to provide outputs at a specified minimum frequency (possibly subject to constraints on the maximum rate of update, and on the regularity of the outputs). These outputs will be required to reflect a sufficiently 'fresh' set of input values. The majority of likely classes of detailed timing requirements arise from these considerations.

3.1 Safety Properties

The majority of properties which do not relate directly to timing are "safety properties". These define behaviors of the system which the controller should never allow. An example might be that

> The EMS should never allow the engine to start while the demanded speed is zero (i.e. the ignition is turned off).

Note that this class of excluded behavior might be sufficient to prevent the system entering a possibly hazardous condition, but that taking action to rectify a potential danger is usually a timing (or liveness) condition rather than a safety property:

> When demanded speed becomes zero, the EMS should bring the engine to a halt in no more than 1s.

3.2 Iteration Rates

The simplest timing requirements are those which state that particular functions (such as providing an output, or updating a state variable) are executed regularly. Such requirements can express a variety of forms of constraint:

- Repetition at a precise period
- Repeated execution within a specified interval of a series of precise intervals. (Placing bounds on the maximum total deviation from an ideal scheduler.)
- Repetition such that the time interval between executions lies within specified bounds. (Placing a constraint on the rate of deviation from the ideal, but not the actual maxmimum displacement.)
- Long-term bounds on average execution rates, placing limits on the number of executions which must occur over some specified time, but not on the actual times of occurrence.

This classification has previously been observed in [7].

Requirements of this form provide one of the most important inputs to the design of a scheduler, and verifying these properties constitutes an important "sanity check" on a proposed design.

3.3 Sequence Timing

A more sophisticated and more stringent requirement is necessary to capture overall response time conditions which are inevitable in embedded systems design. These typically take the form of constraints placed on particular sequences of task executions calls to ensure that the response to a particular input change is made in a timely fashion.

A typical requirement of this form will include a "trigger" task, which is responsible for detecting some input condition, a list of subsequent tasks each of which derives a value from values provided by previous tasks, and a final task which generates the output. An associated time bound will define the maximum allowable delay between input and output, and is thus a maximum bound on the time from the start of the

execution of the first procedure to the completion of the last specified. Any number of other operations could theoretically intervene, and several such conditions, or even several instances of the same condition, can be active at once.

We can model such specifications by considering just the execution of the specified tasks and the passage of time. Suppose we construct a specification process (perhaps modelling an observer watching the system) as follows. The specification process will allow watch for the execution of the first (triggering) task in the sequence, and start a counter when this is observed. It will then wait for each of the required subsequent tasks to be executed (ignoring any irrelevant executions). If the specified execution sequence is completed before the counter reaches the limit, the specification simply returns to its inactive state. If the counter reaches the limit before all the desired executions have completed, the observer should indicicate that the process violates the specification. In the CSP theory, we can actually use such an "observer" as a formal model of the behavior which the system should be permitted to perform. If all the system's behaviors are permitted by the observer, the requirement is satisfied; if not, the system is unacceptable.

We will write the property which enforces this limit as

$$WITHIN(trigger, limit, actions)$$

where *trigger* is the name of the first procedure in the thread, *limit* is the time limit permitted for the thread and *actions* is a sequence of tasks which must be executed, in order, within the given interval. For example, to specify that every change in demanded engine speed is reflected in the calculation of ignition timing within 10ms we might have the condition

$$WITHIN(ReadDemand, 10, \langle ValidateDemand, IgnitionAdvance, DriveIgnition \rangle)$$

In this case we are insisting that every call to *ReadDemand* is followed within an interval of 10ms by the procedures *ValidateDemand*, *IgnitionAdvance*, and *DriveIgnition*, in that order.

The description of the specification above captures the required behavior after a single triggering task. Provided that the initial task does not also form part of the subsequent action, we can allow for overlapping conditions simply by combining several copies of the basic specification process in parallel.

3.4 Conditional Requirements

Many practical systems will be sufficiently complex to need requirements which are enforced only under certain conditions. For example, a controller would probably not need to respond rapidly to inputs if it were in a start-up or self-test mode in which the plant being controlled were guaranteed to be inactive.

This type of requirement can be validated in a number of ways. A simple approach is to explore the behavior of the system in an environment such that only modes in which specific conditions must hold are entered; this may greatly simplify the analysis (whether this is performed by hand or mechanically), and give a significant confidence in the correctness of the schedule. Obviously, however, such partial analysis cannot, in general, be applied to give a formal guarantee of complete correctness.

To establish a formal property, the requirements must relaxed explicitly. Typically the result will be conditional statement such as

> If the engine is currently running (i.e. speed > 100rpm), the EMS should respond to an increase in demanded speed within 10ms.

We should note that significant difficulties have been observed in some cases in defining exactly when informally specified conditions like the one above actually apply: if the engine speed drops below 100rpm 2ms after a demanded increase in speed, is the time limit to apply or not?

3.5 Quantitative Results

The final type of requirements which we might consider as having a direct impact on scheduler correctness are those which place quantitative bounds on the satisfaction of particular deadlines, or on the overall utilization of the system by time-critical tasks.

It is common practice, for example, to insist that a specified proportion of processor time remains free for non-time critical tasks and for possible expansion. (The proportion being higher for early releases of a system than for revised software issues.) Similar considerations, coupled with the possibility of variations in clock frequency and unpredictability in interrupt and IO latency, make it desirable to establish similar margins on other timing constraints.

These conditions are not necessarily to be strictly enforced, however, and thus it can be useful to weaken the original requirement and then measure the relevant timing parameters, such as the minimum delay between a set of real-time tasks completing and the next task being scheduled (for processor slack time) or the interval between the completion of a sequence of desired actions and the latest time at which a particular requirement would have allowed this completion. We can achieve this by associating quantitative information with the specification which enforces the condition, as discussed in Section 5.6 below.

4 Structure of Embedded Programs

Programs for embedded systems of the type we are considering are commonly implemented in a distributed style: the software is structured as a series of distinct tasks, whose execution is triggered by the scheduler.

Inter-task communication is usually implemented by the use of shared variables. As only a single task is executing at any instant, access control is typically not required for these variables[1]. Input and output activities are implemented as tasks which are scheduled in the same way as computation, although it is common practice to arrange that these tasks are executed in close synchronization with a system clock interrupt (by placing them early in each cycle, for example). Failure to make this restriction can result in significant difficulty in ensuring reliable communication, and can produce unacceptable "jitter" in the timing of inputs and outputs.

The scheduler is responsible for executing each task in the system in an appropriate sequence and at specified times. It is common for the sequencing contraints to be represented by a static list of tasks to be executed. A variety of execution rates can be permitted by marking "slower" tasks as being executed only in some cycles of the fastest rate clock, but this scheme does not extend to situations where a low-frequency task may be longer than the available execution time in a high-frequency cycle – in this case, the high-frequency tasks must be able to preempt the low frequency one, and a more sophisticated schedule must be used.

Because of the difficulties involved in preemptive scheduling, it is normally avoided in embedded systems: long low-frequency tasks can be broken in to smaller sub-tasks if required. In this case, a single sequence of tasks may be sufficient to represent the desired operators. The scheduler maintains a counter of the current position of the (highest-frequency) cycle in the (lowest-frequency) series. Tasks are marked with the values of this count for which they are actually active. For example a system with three tasks scheduled harmonically every at frequencies of 1, 2, and 4 Hz might be represented by the following table:

Task	Cycles
A	0,1,2,3
B	0, 2,
C	0

This scheme is simple and easy to implement, but does have a minor potential disadvantage: the order of execution of tasks within a cycle is fixed. For many applications such an assumption is reasonable, as we might expect higher-frequency tasks to require a more stable timing pattern and a lower latency (between the cycle starting and the task executing). If, for some reason, this is not appropriate, we might cast the schedule as a sequence of execution sequences (in the manner of [7]). For the example above we may need to permute tasks A and B, resulting in the following:

[1]The scheduling of inter-processor communication is, of course, the primary issue discussed elsewhere in this project [6]

Cycle	Tasks
0	A, B, C
1	A
2	B, A
3	A

The final complexity to be addressed is the use of application data to change the schedule decisions which are made. Typically, the effect of the application on the scheduler will be restricted, perhaps to the selection of a scheduling mode or some other variable with a restricted domain. If the dependency of a task on such state data is orthogonal to its timing, we can simply express this dependency separately:

Task	Mode		
	Stationary	Accelerate	Decelerate
A	✓	✓	✓
B		✓	✓
C	✓		

If the relationship between timing and system operating state is not orthogonal, we might either (a) provide separate scheduler data for each processor state, or (b) provide different task identities which actually perform the same functional task, but in different circumstances. For example, if task B were required to execute in cycles 0 and 2 when accelerating, but 1 and 3 when decelerating we might introduce a "placeholder" B' which actually performs the same task as B:

Task	Cycles	Mode		
		Stationary	Accelerate	Decelerate
A	0,1,2,3	✓	✓	✓
B	0, 2,		✓	
B'	1, 3,			✓
C	0	✓		

The assumption that B' and B are actually equivalent can be formally represented in a model of the system by renaming any event relating to the execution of B' into the corresponding action of B.

5 Formal Specifications

This section is intended to give formal CSP characterizations of the proerties discussed earlier. Our specifications will be based on the usual notion of *trace refinement*: we give, as a formal description of the property we require, a CSP program which allows all the sequences of events we wish to consider as valid. A scheduling system will satisfy our requirements if all the possible behaviors of the system model are permitted by the relevant specification process.

5.1 Observable events

Our correctness criteria concern the passage of time and the execution of software tasks. We will therefore assume that the following events may be used in representing the behavior of our system:

χ indicates the passage of one unit of time. Our time measurement is discrete, but to investigate the effects of this approximation we can vary the actual units used. To allow a range to be calculated simply and using the integer arithmetic provided by the FDR tool, figures in the model may be written as multiples of some smaller interval; this will allow us to change the accuracy of our model simply be changing a parameter in the scaling function.

exec.i represents the start of execution of a task i. We do not need to model the completion of a task explicitly if it can be inferred to sufficient accuracy for our needs from the start of the next task.

$\lightning$ indicates the start of cycle of scheduler execution. $\lightning$ events are assumed to occur at intervals corresponding to the fastest execution period required by a program. In implementation terms, $\lightning$ might represent a timer interrupt.

5.2 Safety Properties

These form the largest "traditional" class of trace specifications. Note that in order to capture the relationship between scheduling decisions and external events, the actions we observe must be augmented by the events in question. For example, our earlier requirement

> The EMS should never allow the engine to start while the demanded speed is zero (i.e. the ignition is turned off).

can be expressed by the following process (assuming that demanded speed is available frequently on channel *demand*):

```
-- Assuming the initial state is potentially unsafe,
SafetyInitial = SafetyUnsafe

-- This process monitors the condition when the demanded
-- speed is non-zero. It _prohibits_ exec.start.
SafetyUnsafe = demand ? x ->
        if x==0 then SafetySafe else SafetyUnsafe

-- This process monitors the condition when the demanded
-- speed is zero. It _allows_ exec.start.
```

```
SafetySafe =
        demand ? x -> (if x==0 then SafetySafe
                       else SafetyUnsafe)
     [] exec.start -> SafetySafe
```

We then require that our system refines *SafetyInitial*.

5.3 Iteration Rates

The specification of periodic execution has also been the subject of significant work, including work on this project citefunspec. We include these "standard" definitions here for completeness.

If an exact bound is placed on the time between executions, we have

```
PERIODIC(i, T) = |~| t : {0..T-1} @ PERIOD(i,T,t)
```

where i is a task and T is the time required between successive executions. The non-determinsitic choice serves to allow an arbitary starting point to be chosen within the cycle. Once this point is established, the process is deterministic:

```
PERIOD(i,T,n) = if n==0 then
                    exec . i    -> PERIOD(a,T,T)
                    else tock -> PERIOD(a,T,n-1)
```

This keeps a count, n of the time allowed until the next occurrence of *execed.a*. When the count is zero, it will only allow this execution action to occur; when n is non-zero, the process will allow time to pass (represented by the *tock* action) and decrement the count accordingly.

In practice this *PERIODIC* condition is too strong and our specification will allow some variation in execution time. If the requirement places a bound on the allowed deviation or the actual execution and the desired one, we will call it a *bounded drift* requirement, and capture it with the following specification process.

```
BLURRED(i)  =  |~| n : {L_i .. U_i} @ BLUR(i,n)

BLUR(i,n) =
    (if n < T_i + U_i then tock -> BLUR(i,n+1) else STOP)
     []
    (if T_i + L_i <= n then exec.t_i -> BLUR(i,n-T_i)
                       else STOP)
```

The only information that need actually be retained from the history of the process before the last occurrence of the event is the phase shift at which it occurred; this

"drift" (the parameter n) will always lie within the interval $[L_i, U_i]$, and initially we allow it to take any such value.

If the requirement places bounds on the interval between successive executions of a task, rather than on the absolute time of occurrence, we will refer to it as a *bounded rate of drift* condition. The process here is remarkably similar that above; the difference lies in that after the task has been scheduled, its drift is forgotten and the interval counter reinitialized to zero:

```
REPEATING(i) = |~| n : {0 .. Tmax_i} @ REPEAT(i,n)

REPEAT(i,n) =
   (if n < Tmax_i then tock -> REPEAT(i,n+1) else STOP)
    []
   (if Tmin_i <= n then exec.t_i -> REPEAT(i,0) else STOP)
```

Once again the initialization condition can be altered by restricting the range of the non-determinism.

In the weakest form of iteration rate condition, a bound $[Nmin_i \ldots Nmax_i]$ is placed on the number of executions in some time interval $Tint_i$. We need to keep a record of the rate of execution of task i over the last $Tint_i$ time units. This could be held as an integral part of the state of the specification process, as a sequence parameter for example, and the process could count the number of i executions in the sequence before deciding if an execution was permissible (or necessary). Alternatively, we might keep the history in a "delay-line" process, and store the sum as a separate state item. In the following definition, channels *delin* and *delout* are assumed to communicate with such a process.

```
CNTRL(i,sum,curr) =
(if Nmin_i <= sum
    then tock -> delin ! curr ->
                 delout ? v -> CNTRL(i,sum-v,0)
    else STOP)
[]
(if sum < Nmax_i
    then exec.t_i -> CNTRL(i,sum+1,curr+1) else STOP)
```

The initialization of this system is quite important: the initial value of *sum* should equal the sum of the slots in the initial value of the delay-line, and that value should, itself, be between $Nmin_i$ and $Nmax_i$, otherwise *CNTRL* will attempt to bring the execution rate back into line initially (possibly stopping time in the process).

To augment these general specifications, we can employ a valuable practical insight which has arisen in recent work: when checking a refinement automatically, it is usual to insist that the shortest trace leading to an error is returned if the the refinment

fails to hold. In the context of this type of timing requirement, this means that an error is reported when the execution of a task becomes overdue (i.e. when the time limit expires). It is perhaps more useful, however, to give some indication of the interval between a task becoming overdue and its actual execution time. (This is particularly the case with less-critical tasks, where some failure to meet deadlines may be permitted.) Specifications in the above style can provide this information if we change their action on the detection of an error. Most of the processes have a similar form to the following

```
REPEAT(i,n) =
   (if n < Tmax_i then tock -> REPEAT(i,n+1) else STOP)
    []
   (if Tmin_i <= n then exec.t_i -> REPEAT(i,0) else STOP)
```

If the time limit is exceeded, this process will not let χ events occur, and a delinquent implementation will typically fail to refine this (because it will allow χ but not the scheduling *exec.* action). If we rewrote the specification including

```
REPEAT(i,n) =
   (if n < Tmax_i then tock -> REPEAT(i,n+1) else IDLE)
    []
   (if Tmin_i <= n then exec.t_i -> REPEAT(i,0) else STOP)
```

where

```
IDLE = tock -> IDLE
```

then the refinment would not fail until the implmentation was willing to perform the *exec.* action (which the specification would no longer permit). The number of χ events which had elapsed since the deadline now provides a measure of the time delay in meeting the deadline.

We should perhaps note that this simple form actually permits a refinement to hold if the implementation never schedules the task being considered. A stronger condition could employ a scheme like the following:

```
REPEAT(i,n) =
   (if n < Tmax_i then tock -> REPEAT(i,n+1)
                  else IDLEFOR(2 * Tmax_i))
    []
   (if Tmin_i <= n then exec.t_i -> REPEAT(i,0) else STOP)
```

where

```
IDLEFOR(n) = if 0 <= n then tock -> IDLEFOR(n-1) else STOP
```

This specification will prohibit any behavior with more than $2Tmax_i$ χ events after a deadline has been missed, as well as attempts to execute the task once the deadline has expired. Checking refinement of such a specification can have one of three outcomes:

- Successful refinement, indicating that the deadline is always met,
- A failure resulting from the implementation allowing an *exec.* event which the specification did not permit. This indicates that a deadline has been missed, and allows the amount by which it was missed to be determined from the trace leading to the error.
- A failure resulting from the implementation allowing a clock tick (χ) which the specification did not permit. This indicates that the implementation missed the deadline by more than the amount specified ($2Tmax_i$ in the above example).

5.4 Sequence Timing

The description of timing requirements applicable to sequences of task executions given in Section 3.3 was already phrased in terms of the actions which might be seen by an observer. To encode these requirements in a form suitable for verification by refinement, we simply need to express such a non-determinsitic observer in CSP.

We first define some processes used in constructing our specifications. Each specification will consist of two parts, one which keeps track of the elapsed time and another to monitor the calls to specified procedures inorder. *reset* is the event used by the latter to inform the timer that the thread has been successfully executed. *synch* is a further synchronization introduced to ensure that the processes remain "in step".

```
channel reset, synch
```

The events *irrelevant* and *therest* are used in the following definitions as short-hand for procedure calls which do not satisfy the thread.

```
channel irrelevant, therest
```

Limit is the timer half of the specification – when proecdure *trig* is called, it enters the *Bound* state and states counting down the number of *tock* events allowed by the *limit*.

```
Limit(trig,count) = tock -> Limit(trig,count)
                    []
                    execed.trig -> Bound(trig,count,count)
                    []
                    therest -> Limit(trig,count)
```

Bound decrements the timer value whenever *tock* is observed, and simply ignores procedure calls. When a reset occurs, it makes a further synchronization on channel *margin* (this is used in later specifications to allow timing margins to be examined), and then resets to its initial state. If the time limit is exceeded, further *tock* events are not permitted. (This will be visible in practice as a failure of the refinement check.)

```
Bound(trig,count,curr) =
 (if 0 < curr then tock -> Bound(trig,count,curr-1)
             else STOP)
  []
  reset -> (synch -> Limit(trig,count)
         [] margin ! MarginQuantum * (curr/MarginQuantum) ->
                   synch -> Limit(trig,count))
  [] therest -> Bound(trig,count,curr)
  [] execed ! trig -> Bound(trig,count,curr)
```

The basic unit used to build the part of the specification which monitors procedure executions is $Await(f, X)$. This will perform any event from the set X, and either terminate successfully if the event is f, or remain in the same state if not.

```
Await(f,X) =
  ([] x : X  @ x -> if x == f then SKIP else Await(f,X))
```

The following definition shows how these elements may be combined. We define a process (*WithinEg*) which specifies that each of the tasks in *actions* must occur (in order) within *Tlim* of any execution of *trigger*.

We instantiate a copy of the *Limit* process together with a sequence of *Await* processes which check for each required procedure in turn. This sequence terminates (and loops back to its initial state) only when all the procedures have been called in order.

```
WithinEg = ( Limit(trigger,Tlim)
        [[ therest <- exec.head(actions) ]]
  [| union({| exec.i | i<-{trigger,head(actions)}|},
           {| reset,synch |}) |]
     while(rseq(y,<exec.trigger>^
                  <execed.i | i<-actions>^
                  <reset>,
     if y == reset then
      [] x:{exec.i,irrelevant |
            i<-union({trigger},set(actions))} @
          x -> reset -> synch -> SKIP
     else
```

```
    Await(y, union({irrelevant},
       { execed.i | i<-union({trigger},set(actions)) }))))
  ) \ {reset,synch}
```

The *reset* and *synch* events are not intended to be part of the specification, and thus are hidden from the environment.

To use such a specification, we take a model of our system, and map task executions which do not concern the particular requirement to the *irrelevant* value introduced above. We may then hide any other events which do not directly appear in our specification, such as the *cycle* event marking the beginning of a frame.

```
TestEg =
    (System [[ execed.i <- irrelevant
             | i <- diff(TASKID,
                    union({trigger},set(actions))) ]])
    \ {| cycle |}
```

The condition we require is

```
assert WithinEg [T= TestEg
```

We should note that the above specification process makes two assumptions:

- No more than on instance of any sequence is "active" at any one time, and,
- the triggering event of a sequence does not also occur in the list of triggered actions.

The first of these is particularly restrictive, but is easily relaxed by expanding our specification. Rather than simply comparing the implementation to a single process like *WithinEg*, we may compose several specifications in such a way that they synchronize on timing signals and task execution other than the trigger, but interleave on the triggering event itself. This allows successive occurrences of the triggering task to start different counter processes, and enforces the timing condition on each. In practice, the number of overlapping threads of this sort seems not to be large, and thus we do not need to compose many concurrent observation processes.

The second restriction appears less of a practical problem, and could be similarly removed, although possibly at the cost of adding a further parallel process to control the distribution of execution signals.

5.5 Conditional Requirements

Where a condition is required to be enforced in a limited range of system stages, it is obviously necessary to be able to identify when the process enters or leaves such

states. This will require that our implementation model be extended to offer additional communications specifying the critical states. In many cases, the information will be easy to identify in a straightforward model of the software; the model of a task which examines operating conditions and changes the system mode accordingly can easily be extended to signal such changes on an extra channel. An alternative approach might add a monitor in parallel with the implementation (or the specification) to detect and signal changes in more complex conditions.

Suppose, then, that this enables us to provide a model which engages in an event from set R_I when entering a state where some trace condition should be enforced, and an event from set R_O when leaving such states. If the requirement would normally be enforced by a process *SPEC*, we may construct a process which allows behaviors of *SPEC* between R_O and R_O, but arbitrary traces otherwise using the CSP interrupt operator as follows.

```
SPEC' = ([] x:R_I @ x -> SPECactive) []
        ([] x:R_O @ x -> SPECinactive)

SPECactive = (SPEC ||| RUN(R_I)) /\
             ([] x:R_O @ x -> SPECinactive)

SPECinactive = RUN(diff(SIGMA,R_I)) /\
               ([] x:R_I @ x -> SPECactive)
```

(This assumes that an event from $R_I \cup R_O$ will precede any other to determine the initial state – other initial assumption are clearly trivial to encode.)

The above definition assumes that the *SPEC* condition is independent of $R_I \cup R_O$ and may be re-initialized whenever it becomes necessary to enforce it. Other conditions can be captured by, for example, maintaining some separate specification state in a process placed in parallel with *SPEC'*, or placing *SPEC* in parallel with a process which will only allow it to progress between R_I events and R_O events, and otherwise permits the implementation arbitary trace behavior[2].

5.6 Quantitative Results

Our specification for a sequence timing constraint is constructed as an observer, monitoring the tasks which are executed and maintaining a count of the time elapsed since the start of a sequence. The value of this counter at the point where the execution of the last task in the sequence is observed may clearly be used to give a measure of the margin remaining before the deadline in a particular point in the execution. Every

[2]This will obviously entail renaming the entire alphabet of *SPEC* in order to interleave two alternative behaviors, and renaming them back to the implementation view at the outermost level.

state of the specification at in which the timing requirement has just been satisfied will have such a value associateed with it. Thus if we can identify the set of states which our "observer" can reach while observing our system, we can find the set of possible margin values, and in particular we can identify the least.

When we prove that a refinement holds using **FDR**, the tool builds a set of pairs relating each state in the implementation of our system to the corresponding states of the specification which it satisfies. The set of states which the observer can reach, therefore, is simply the set of states which are related to one or more states of the implementation by a *successful* **FDR** check. (If the check is not successful, the complete state space will not have been explored, and consequently the approach we are outlining does not hold – we might not expect to be able to measure a safety margin in an unsafe system!) A small modification to the refinement-checking program used in **FDR** easily allows us to determine which states of the specification were visited in the course of demonstrating a successful traces refinement, but this information is initially in a form which refers only to the observable behavior of the specification, and in particular it refers to a process which has been compressed in the course of normalization[3].

We thus must solve two technical difficulties to use the technique in practice: we must prevent **FDR**'s compression algorithm from merging states with the same observable sequences of behavior but different time margins, and we must provide a way of encoding this timing information in a way which allows it to be retrieved from the compressed machine. A simple matter of CSP programming addresses both issues: we add an additional communication channel, *margin* say, to the specification process, and modify the observer to *permit* an output of the remaining time whenever the last event in a sequence is observed. We do not make this event compulsory, however, and its occurence need make no change to subsequent behavior. This addition does not prevent any process which previously satisfied the specification from continuing to do so, because the additional communication is always merely a choice which the implementation can chose to ignore[4]. As the only additional events are confined to a channel whose name can be chosen so as not to appear in the implementation model, this change equally does not accept any otherwise prohibited implementations[5].

The additional possible outputs of distinct values prevent the normalization from identifying states with differing margins (although the effect on the size of the whole specification will only be additive in the case that immediately following the comple-

[3]Version 1 of **FDR** uses a strong bisimulation relationship to factor the pre-normal form. **FDR 2** incorporates a wide range of semantic compressions.

[4]The nature of the choice (determinsitic or nondeterministic) is irrelevant for proofs of *traces* refinement.

[5]Formally, we insist that the new specification (*Spec'*) relates to the old as follows: $Spec =_T Spec' \setminus \{margin\}$ and so $Spec \sqsubseteq_T P$ exactly when $Spec' \setminus \{margin\} \sqsubseteq_T P$, and thus as $P \setminus \{margin\} = P$, $Spec \sqsubseteq_T P$ when $Spec' \sqsubseteq_T P$.

tion of a sequence the observer was to return to a "reset" state), and also provide a convenient means of presenting the state information to the user. The modified **FDR** returns the union of all the possible initial events of states visited by the specification. A timing margin of t was possible exactly when $margin.t$ appears in this set. Note that we must use the set of possible actions from each specification state, as we guarantee that a transition $margin.t$ is never performed by the implementation, and thus never explored further by the refinement engine.

Let us now consider the similar problem of measuring processor slack time in cyclicly scheduled system. The tasks associated with each frame will be executed when the frame starts, and assuming that our design is adequate, these will terminate some time before the next frame is due to start[6]. In practice, the time between one frame and the next will be occupied with non-critial background tasks. In a similar manner, we may add to our formal model a task which is active over this interval and which simply counts the time periods which elapse until the next frame start signal is actually generated – it is usually straightforward to do this in a way which does not change the behavior of the overall system with respect to the original interface of the model. If this task is able communicate these time values to the environment on some channel, the possible values of processor slack time can simply be extracted by examining all possible behaviors and noting the values transmitted on this channel. As this channel is an artifact of our modelling rather than a physical entity, our specifications can simply be expanded to ignore it:

$$SPEC' \quad = \quad SPEC \;|||\; CHAOS_{\{slack\}}$$

If a finite-state process communicates only finite ranges of values over a finite set of channels, the set of events which are actually possible can be calculated by searching the space of possible behaviors – modified versions of **FDR** and a simple program based on the **FDR 2** library have already been constructed for this task.

6 Modelling the System

As noted above, the programs we consider consist of a number of top level tasks executed in sequence by a scheduler procedure which is itself executed at regular intervals by the run-time system. Each excution of the scheduling procedure is a *frame.* The entire scheduling sequence will typically repeat at a somewhat lower frequency – each such complete set of frames will be termed a *cycle.* Less critical tasks may, in turn, be run according to a sub-schedule only once in a number of cycles (the possibility of abstracting from the details of these infrequent tasks will be discusse below. Additional non-time critical (background) tasks may in turn be

[6]Assuming, for the present, that our real-time tasks are not pre-emptable.

executed when the processor is idle – these tasks are not considered further in this report.

The major components of the formal model represent functions which can be identified in the actual software structure of such a program:

- A timer process which relates the occurrence of $\frac{1}{2}$ events (indicating the start of a frame) to the physical passage of time, and thus is an analogue of the hardware timer and run-time system in an actual system. Our formal model will simply count the number of time intervals which are permitted before the next interrupt, and allow time to pass (if this number is greater than zero), or signals the interrupt (if the number is equal to zero). The following definition uses *tock* to represent the passage of time, and *cycle* to represent the interrupt.

```
Timer(n) = Timing(n,0)

Timing(n,m) = if   m == 0
                   then cycle -> Timing(n,n)
                   else tock -> Timing(n,m-1)
```

  The variable m keeps track of the time to the next interrupt, and n represents the cycle length which is used to reset m when an interrupt actually occurs.

- A "store" process will keep track of the information which the scheduler uses to determine control flow. This will include both scheduler specific state, such as the frame and cycle counters, and any application-dependant data (such knowledge of any failures observed in a fault-tolerant network, system operating mode, etc.). Counters may be modelled simply as a process which maintains a number which is incremented periodically and which may be read by the scheduler when required. In modelling other state information, we may take any conservative non-determinstic approximation to the actual system behavior and still be assured of the validity of our analysis. Indeed, if no constraints are placed on the systems behavior across state boundaries, an entirely arbitrary selection of mode data will be permissible.

- The most complex process is that which represents the control flow through the actual program. In its simplest form, this may closely resemble the procedural code for the scheduler implementation, taking the form of a loop which indexes a data table including information about each task. The actual style of representation will be discussed below.

6.1 Representing task data

There appear to be two distinct possible methods of representing the task set present on a cyclic-scheduled system, one using parallel composition, and the other effectively sequentializing the information.

Perhaps the most natural respresentation involves defining a concurrent process for each task, and placing all these processes in parallel with a scheduling process which ensures that no more than one task executes at any given time. (This approach is demonstrated in [8], for example.) A typical task might take the form

```
TASK(id, duration) =
        exec.id -> Running(id,duration,duration)
        []
        tock -> TASK(id,duration)

Running(id,maxduration,remaining) =
    if remaining == 0 then
        done -> TASK(id, maxduration)
    else
       (done -> TASK(id, maxduration) |~|
        tock -> Running(id,maxduration, remaining -1))
```

The task switches from idle to running on recipt of an *exec* signal, and executes for a maximum of *duration* time units. On completion it informs the scheduler via the *done* signal and returns to the idle state. Not only is this approach close to the conceptual model used in designing the system, but it permits processes to maintain internal state if this is desirable. For example, a task which dispatched according to a sub-schedule could maintain a variable recording its position in that schedule, or the model of a communication task might be permitted to assume (from external data-rate considerations) that a buffer could not be full to its maximum extent on two adjacent calls. As regards formal manipulation, however, this representation does have some drawbacks; in particular the assumption that at most one process executes at a time must be enforced external to the definition of the task set (by the scheduler)[7].

An alternative approach which makes the fact that only a single process is active at any one time explicit is to represent the execution of tasks as a sequence of process invocations. A typical task might now be represented

```
TASK(i, d) = exec ! i -> RUNNING(d)
RUNNING(n) = if   n == 0
```

[7]The resulting inefficiency of state representation, and particularly the inefficient representation of multi-way synchronization, are major deficiencies in the FDR-1 tool in this context.

```
then SKIP
else (SKIP |~| tock -> RUNNING(n-1))
```

The *exec* signal now serves only to inform the environment of teh execution of the task (for specification purposes). When a task completes, it simply terminates successfully. The complete task set is now represented as a sequence of tasks:

$$TaskSet \quad = \quad Task(A, 30) \,;\, Task(B, 45) \,;\, \ldots \,;\, TaskSet$$

(Note that recursion or iteration can be used to complete the cycle at the end of a single pass.) Additional processes may be added to the sequence to model the interaction of sequential behavior with cycle interrupts, and to model processor slack time (or background processing). State values, however, must be maintained by a separate parallel process, as the CSP sequential composition operator does not transfer internal state.

In either of the above cases, the simple task model given above can easilt be extended to allow for conditional execution according to a value provided by a cycle count or state variable process.

To avoid the complexity of modelling long cycles, we can abstract away from the details of some infrequent tasks. Suppose the highest frequency tasks in our system were executed every 5ms (the frame time), and the majority of tasks were repeated every 50ms (i.e. every 10 frames). If one of these tasks in turn executed a sub-schedule and invoked relatively infrequent (period 1s, say) tasks, then a full inductive proof (or mechanical analysis) would have to consider at least a possible (1000ms/5ms =) 200 combinations. In many cases, however, the time occupied by the infrequent tasks will be small, and we may make the *conservative* assumption that every invocation of the sub-scheduling task occupies a non-determinsitic amount of time up to the actual worst case. If this approximation can be made, we need only consider (50ms/5ms =) 10 configurations. It may be, of course, that the schedule will not meet all its requirements under this assumption, but we can be sure that those it does meet certainly hold of the actual system.

7 Verification

The essence of verifying that a model of the form outlined above satisfies specifications as discussed in Section 5 is to prove that a refinement relation holds between the CSP specification process and the implementation model. It is important to note that although many of the requirements discussed would conventionally be thought of as liveness properties (tasks *will* be executed within bounds), the addition of quantitiative timing constraints results in a property which can be expressed as a condition on allowable traces, coupled with the necessary proof that the implementation does not

"stop time" by entering a state after which no timing events are possible or by insisting that infinite computation occurs between timing events. We may check these latter "well-formedness" conditions by verifying a single failures-divergence refinement:

$$RUN_{\{tock\}} \quad \sqsubseteq_{FD} \quad System \setminus Sigma - \{tock\}$$

Once this is established, timing properties can be verified by considering only the traces of the processes involved:

$$Spec \quad \sqsubseteq_T \quad System$$

Because the CSP sychnronizing parallel operator ($[\![X \mid Y]\!]$) can be used to represent conjunction in the traces model, we may prove satisfaction of each of our timing properties separately. In addition, in some circumstances we can exploit the following law

$$S \sqsubseteq_T P \quad \Rightarrow \quad S \sqsubseteq_T P \ [\![X \mid Y]\!] \ Q$$

to "factor out" that part of our system model which enforces a particular condition and thus simplify our analysis further.

The relationship between trace refinement and parallel composition can be exploited further if we need to verify a condition which depends on assumptions about the environment: placing a deterministic process which enforces the condition in parallel with our system model will constrain the space of possible behaviors accordingly. We should note, however, that in order for this approach to be valid, the "well-formedness" property above must be shown to hold of the system and constraint together. (Failure of this condition implies that the system and constraint are in fact inconsistent.[8])

Another application of parallel composition is the addition of details about tasks which appear to the scheduling system as a single entity, but which implement a sub-scheduled sequence of actions internally. For example, an adaptive filtering task may need to be scheduled every frame (in order to accept inputs and provide filtered output), but might contain parameters which should only be updated relatively infrequently. If this activity is split into two separate tasks, the overall number of items to be run increases (consequently increasing overheads), and the abstraction of the "filter" is broken – the updating task must have access to what could otherwise be private data.

We may instead decide to allow the filter to perform the updates, using an internal counter to keep track of when this is required. This internal detail need not be made explicit in the overall model unless the updating process takes sufficiently long that it does need to be considered in the top-level schedule, in which case, of course the

[8]If the constraint is deterministic.

abstraction must necessarily be broken. To verify that the filter process as a whole is scheduled sufficiently often, the model need only refer to single event *exec.filter*. To guarantee properties of the update, however, we may constuct a separate model of the internal state of the filter:

```
Filter(n) =     tock -> Filter
              []
                exec.filter -> if n == 0 then
                        exec.update -> tock -> Filter(N)
                else Filter(n-1)
```

We include explicit reference to the timing event here in order to insist that the exec.update is made visible to the environment as soon as it becomes available. (The priority mechanism discussed in [2] will provide a more satisfactory solution to this type of problem.)

We now check

$$Spec_{Update} \sqsubseteq_T Filter(N) \,[\![\, \{exec.filter, tock\} \,]\!]\, System$$

Because the *Filter* process has no impact on the operation of other tasks, however, we need not treat it as an integral part of the model, and need only include it when testing specifications that reference *exec.update*.

Assuming that the refinement relations which need to be established are being verified mechically by the **FDR** tool, extraction of the additional information required to give quantitative results is straightforward, and in the majority of cases causes no significant run-time penalties. As discussed earlier, the quantitative measure associated with a specification state can be encoded in the set of events which the state may permit (its initials). A simple modification to the **FDR** system allows the refinement checking process to record which specification states were visited in the course of a check, and to export this information at the end of a check. The current application only requires that the union of the initials of each visited normal-form state is produced. We then may examine which events on any nominated channel were possible, and so deduce the possible values of the timing margin (possibly rounded to some degree). Processor slack time can be similarly extracted from the set of all events occuring in any trace of the implementation, either by modifying the current **FDR** refinement engine, or by using a specialized program using the **FDR 2** libraries.

In either of these cases, of course, we are most interested in the minimum values of the sets of (margin or slack) times returned by the modified tool. It is perhaps interesting to notice, however, that the profile of possible timing margins does provide some information: if a particular timing requirement is initiated by a frequent event and ultimately discharged by the occurrence of a task which is executed less often, the observed margins will "cluster" according to the number of frames which can elapse between the execution of the tasks.

8 Summary & Implications

The overall conclusions of our work on a number of examples of the form described here are encouraging: we have used this approach on practical problems taken from important application areas, and have had little difficulty in obtaining useful results. In particular, even "real-world" schedulers produced without the intention of applying a formal model seem amenable to capture in CSP, and the models that result are not overly complex for analysis using existing tools.

The one parameter which is most critical in determining the overall size of such models is the time unit chosen as the basis of our discrete-time models. By using a scaling function whenever time values appear in our models, we are able to adjust this parameter easily and so gain confidence in the stability of our results. In our largest example to date, the base time interval could be changed by more than an order of magnitude without any observable change in behavior (other than the inevitable variation in rounding).

Two features of the problem domain seem to simplify the analysis of these systems. The first relates to the requirements placed on them: the majority of the constraints which an embedded system is required to satisfy fall into a small number of clearly identifiable catagories, including

- Safety and data-dependence properties (limiting the permissible sequences of actions).
- Iteration rate and timing properties (specifying when a particular task must be invoked).
- Task sequence constraints (specifying overall timing constraints on particular computation sequences).

This regularity means that a great many requirements can be expressed in terms of a few basic definitions. The ability of the theory to support compositional verification means that checking these properties is then a mechanical task, involving little human input once the data entry is complete.

A similar regularity appears in the model of the actual implementation of the schedule. The structure of the model is essentially fixed by the scheduling strategy and the architecture of the implementation. Once this has been captured, subsequent changes to the set of tasks executed or their ordering only involve changes to the small number of data-structures required to define a parallel or sequential set of processes as discussed in Section 6.1. A particularly important feature is that only a single core model of the implementation is required to verify a wide range of conditions and extract a number of timing measures.

There are obviously some issues which must be addressed before the techniques are suitable for wide-spread use. Principal among these are the performance limitations

of the current production versions of the **FDR** tool in systems involving large multi-way synchronizations[9], and the difficulty in maintaining models of (potentially rapidly changing) systems without large amounts of effort by skilled personnel. Integration of the methods described for extracting timing margins and processor utilization into the supported interface of the tool is also obviously desirable. Extension of this style of analysis to pre-emptive systems (following, perhaps the trivial example in [8]) would address the concerns of a wider application group than the non-pre-emptive examples analyzed to date.

Other areas which also deserve investigation in the long term include limiting the effects of the use of a discrete-time model, possibly including the use of symbolic manipulation of continuous values.

Acknowledgements

The work outlined in this paper is a summary of the development of techniques using CSP and **FDR** over a number of years, and in collaboration with a number of groups. Specific thanks are due to: Mike Bardill, Stuart Dootson and Peter Summers of Rolls-Royce Aerospace; Eddie Williams and Paul Jackson of Rolls-Royce & Associates; Neil Brock, Rick Harper, Beth Stamford and Sharon Donald of C.S. Draper Laboratories; Janet Barnes of Smiths Industries; and colleagues at both Formal Systems companies.

References

[1] N.A. Brock. Real-Time Scheduler: Natural Language Problem Statement. Technical report, Charles Stark Draper Laboratory, Inc., 1994. Deliverable D2.1 of SBIR N00014-93-C-0213, in [3].

[2] M.H. Goldsmith. A CSP Priority Operator for **FDR 2**; Prototype Software for Discrete Real-time Extensions to **FDR**. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [5].

[3] M.H. Goldsmith et al. *N00014-93-C-0213: Second Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[4] M.H. Goldsmith et al. *N00014-93-C-0213: Third Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[9] **FDR 2** has been used on models of the sort described here with considerable success, as described elsewhere.

[5] M.H. Goldsmith et al. *N00014-93-C-0213: Fourth Quarterly Report.* Technical report, Formal Systems Design and Development, Inc., P.O. Box 3004, Auburn, AL 36831-3004, 1994.

[6] David M. Jackson and Richard O. Chapman. Models of the Fault-Tolerant Processor; Architecture and Verification. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverable to SBIR N00014-93-C-0213, in [5].

[7] David M. Jackson and M.H. Goldsmith. Specifying Task Management; Single Processor Systems. Technical report, Formal Systems Design & Development, Inc., 1994. Deliverables D 2.2 and D 2.3 of SBIR N00014-93-C-0213, in [4].

[8] D.M. Jackson and M.H. Goldsmith. Scheduler Specification in CSP: Fixed Priority Pre-emptive Example. Working Paper W.2.2.1, Formal Systems Design & Development, Inc., April 1994. Included in Report of SBIR N00014-93-C-0213, in [3].

A Example – Auto Engine Managment System

A.1 Interfaces

The engine management system reads the following values: [Recall that we intend the following example more as an example of task mix, timing and dependency than as a practical system design!]

- Engine speed
- Accelerator pedal angle
- Cooling water temperature
- Oil temperature
- Fuel pressure
- Exhaust gas data

and provides the following outputs

This is Release 2.0 of this document, last modified by Michael Goldsmith at 20:34:42 GMT on April 25, 1995.
This is Release 2.0 of this and the following sections, last modified by Michael Goldsmith at 20:37:41 GMT on April 25, 1995.

- Injector timing required
- Ignition timing required
- Fuel pump drive
- Water pump drive
- Tachometer drive
- Engine status indication

The highest frequency outputs should be recalculated at a frequency of 160Hz; more slowly changing signals should be monitored and/or recalculated at 80Hz or 40Hz as outlined below[10]

A.2 Tasks

The system is implemented by the following tasks:

Acronym	Function	Frequency/Hz	Max duration/μs
RAA	Read Accelerator Angle	80	300
RSD	Read SpeeD	160	500
RFP	Read Fuel Pressure	160	300
ROT	Read Oil Temperature	40	250
CSD	Calculate Speed Demand	80	1000
CIT	Calculate Injector Timing	160	700
CFP	Check Fuel Pressure	160	300
COT	Check Oil Temperature	40	250
DI	Drive Injector	160	500
AGT	Adjust iGnition Timing	80	800
DFP	Drive Fuel Pump	160	300
RXA	Read eXhaust Analysis	40	400
AMX	Adjust MiXture target	40	400
RWT	Read Water Temperature	40	250
CWT	Check Water Temperature	40	250
DCP	Drive Cooling Pump	40	300
LSS	Limit Speed Schedule	40	400
IES	Indicate Engine Status	40	800
DTM	Drive TachoMeter	40	250

[10] 160Hz ensures at least one update per revolution up to a "red-line" of 9600rpm.

This task set is perhaps somewhat small by typical application standards, but not unrealistically so. The maximum durations are hypothetical, but again, we feel that they are not unreasonable. In a practical system there may be a greater range of times than 4:1, but many real tasks can be achieved in less that 250μs, and adding smaller tasks does not in general complicate the scheduling problem as much as adding large ones!

We will assume the data dependency relationships between these tasks is as shown in Figure 1 (over a four frame cycle).

A.3 The Schedule

The following dependencies exist between tasks *within* a frame:

Precedes		Precedes	
RXA	AMX	RWT	CWT
CWT	DCP	CIT	AGT
RAA	CSD	ROT	COT
COT	CSD	CIT	CFP
CIT	DI	RFP	CFP
CFP	DFP	RSD	DTM
RSD	CIT	CSD	CIT

We can use this information to derive a permissible ordering for tasks within a frame mechanically (using the Unix tsort(1) utility, for example).

Adding in those tasks which have no dependencies within a frame, we propose to execute the EMS tasks on a four frame cycle according to the following schedule:

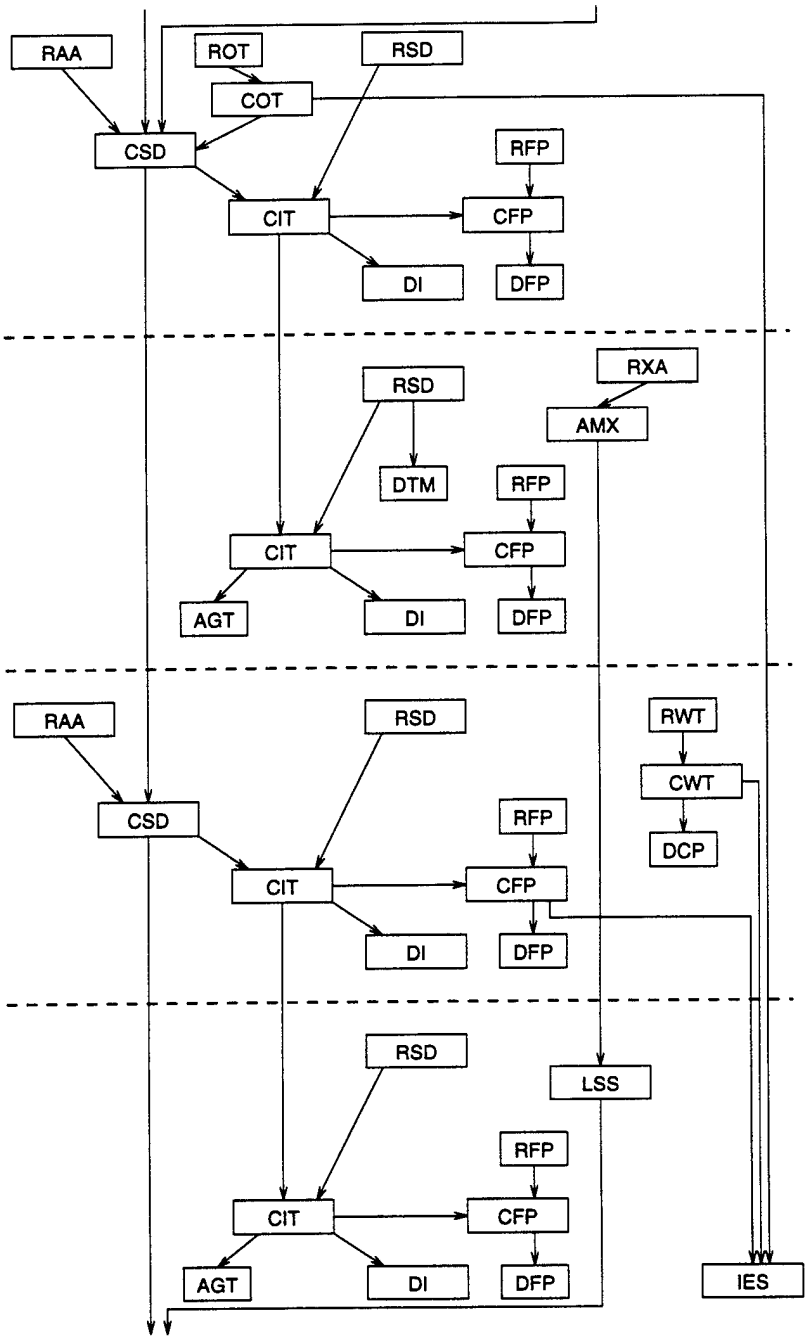


Figure 1: Engine Management System Data-Flow Diagram

Relative order	Task Identity	Active in Cycles			
1	RSD	0	1	2	3
2	RFP	0	1	2	3
3	ROT	0			
4	RAA	0		2	
5	RWT			2	
6	RXA		1		
7	DTM		1		
8	COT	0			
9	CSD	0		2	
10	CIT	0	1	2	3
11	CWT			2	
12	AMX		1		
13	DI	0	1	2	3
14	CFP	0	1	2	3
15	AGT		1		3
16	DCP			2	
17	DFP	0	1	2	3
18	LSS				3
19	IES				3

A.4 Results

A.4.1 Well-formedness and Processor Utilization

The model presented above is well-formed (implying that no over-runs occur). Using a (rather coarse) 100μs time step, the model contains approximately 20000 states. The observed processor slack is as follows:

Cycle	Slack/μs
0	1 700
1	1 900
2	1 800
3	1 600

(These figures indicate an average processor utilization of around 70%.) For a time step of 50μs, these figures became 1.6ms, 1.9ms, 1.8ms and 1.6ms, extracted from a model with around 67000 states, indicating that the results are relatively independent of the time-step chosen provided it is not significantly larger than these figures[11].

[11]It is useful to observe that it does not appear practical to model using time-steps smaller than 50μs with version 1 of **FDR**: with a 20μs step, we were unable to compile the model within the 200Mb available on our SparcStation. Using **FDR 2**, however, the compilation phase was negligible, and the check completed after exploring (the relatively small number of) 385304 states.

A.4.2 Iteration Rates

The iteration rate of each task was checked against a specification insisting that an execution of each task must occur at some point within the time interval allocated to each frame in which the task is due to execute. If t represents the iteration rate of the task in frames ($t \in \{1, 2, 4\}$), and p is the number of the first frame in which the task executes (representing the "phase": $p \in \{0, 1, 2, 3\}$), these intervals can be expressed as

$$[6250(tn) + 6250p, 6250(tn) + 6250(p + 1)]$$

for increasing integer values of n.

While the design we have given above certainly meets all the iteration requirements, one minor difficulty was encountered in the verification which casts some light on both the construction of models and on the practical difficulties of satisfying this form of specification. Our discrete time model necessarily uses integer arithmetic to represent time values, with the consequence that if a time-step (such as 200μs) is chosen which does not divide exactly into the frame length (6250μs), rounding errors will be introduced: a frame will be assumed to be 31 clock cycles in duration. If the specification defines periods using the same frame length value, the rounding will be consistent in both specification and design, and the verification will proceed without difficulty. If, however, the requirements are expressed in different numerical terms (for example as a period of 25ms) it may be represented exactly (as 125 cycles). The implementation will have a cycle length of only $4 * 31 = 124$ cycles, however, and thus will ultimately move out of step with the specification: **FDR** does indeed show this happening[12]. As a modelling issue, then, we propose that the time-step should be taken as an exact divisor of the frame (and thus the cycle) length whenever possible.

As a more practical issue, this kind of drift shows that with this kind of analysis we can at best hope to demonstrate correctness with respect to the system clock: if we wish to include the effect of clock drift or uncertainty we must include these effects in the model explicitly (perhaps by weakening the definition of the process which relates cycle interrupts to clock events).

Input-Output Timing

For the purposes of this example, consider the following set of execution sequences which may be the subject of end-to-end timing constraints:

[12] And as the difference in cycles is small ($< 1\%$) the effort required to detect the error is large – up to 100 times the size of a successful check.

Identifier	Overall limit/μs	Task sequence
a	6250	⟨ RFP, CFP, DFP ⟩
b	6250	⟨ RAA, CSD, CIT, DI ⟩
c	25000	⟨ RXA, AMX, LSS, CSD, CIT, DI ⟩
d	6250	⟨ RSD, CIT, DI ⟩
e	12500	⟨ RWT, CWT, IES ⟩
f	25000	⟨ RFP, CFP, IES ⟩
g	6250	⟨ RWT, CWT, DCP ⟩
h	6250	⟨ ROT, COT, CSD ⟩
i	25000	⟨ ROT, COT, IES ⟩
j	25000	⟨ RSD, DTM ⟩
k	12500	⟨ RAA, CSD, CIT, AGT ⟩
l	12500	⟨ RSD, CIT, AGT ⟩
m	12500	⟨ ROT, COT, CSD, CIT, AGT ⟩
n	25000	⟨ RXA, AMX, LSS, CSD, CIT, AGT ⟩
n'	31250	⟨ RXA, AMX, LSS, CSD, CIT, AGT ⟩

Comments justifying the choice of these sequences are given in the **FDR** input listing below. Note that tests n and n' are included to show that execution sequences can extended beyond the length of a single cycle: requirement n is not met by our system, but n' is.

We can provide a CSP specification which insists that whenever the first task in such a sequence is executed, the remaining tasks must follow, in order, within the specified time. Out-of-order or duplicate executions are ignored. The key components of the specification process are

- A *Limit* process which observes occurrences of the triggering task, and causes an error if the time bound subsequently elapses without a signal being permitted by
- A sequencing process which waits for each element of the sequence in turn, and resets the counter when all have been observed.

Using the **FDR** input scripts included below we can show that our system satisfies all the requirements a–m and n'.

Furthermore, we can extend the *Limit* process to output a measure the the timing margin remaining when the sequence is successfully completed. Using a time-step of 250μs, we obtain the following values:

Identifier	Overall limit/μs	Range of margin	
a	6250	0–500	3000–6250
b	6250	3000–6250	
c	25000	2250–28	
d	6250	2250–6250	
e	12500	0–1000	
f	25000	0–500	
g	6250	0–6250	
h	6250	3750–6250	
i	25000	0–750	
j	25000	4000–6250	16500–18750
k	12500	0–7250	
l	12500	0–6250	
m	12500	0–7000	
n	25000	none!	
n'	31250	0–7000	

One obvious feature of these figures is that several threads have minimum margins of zero – a consequence of our artificial example which would be exceedly worrying in practice! In any practical situation, of course, our timing requirements are unlikely to correspond so closely with the design which implements them. If we permit an additional 1ms delay in each thread, for example, these zero margins are completely absent.

We can also observe that some execution paths yield two distinct ranges of times; these will correspond to different sequences of execution which exhibit the task sequence in question. For example, requirement j requires than an exeution of DTM follows each RSD within 25ms; DTM is executed only once in each cycle and thus there may be 0, 1, 2, or 3 frames between a call of RSD and the next DTM. (We may see fewer distinct ranges as these cases may yield overlapping margins.)

A.5 Summary

This simple, though we feel not unrealistic, example shows how formal specification can be used to verify four key properties of a cyclic embedded system scheduler:

- Absence of overrun; all tasks in one frame are complete before the next frame interrupt occurs. (Overruns would be manifest as deadlocks in which the clock process expected an interrupt to occur while the task execution processes did not.)

- Processor slack times (contingency); we are able to determine approximate numeric values for the number of idle clock cycles which can be guaranteed to be

available in each frame, under the most pessimistic interpretation of our timing data.

- Iteration rates and phases; relatively straightforward tests can demonstrate that each task is executed in the time allocated to the frames in whichit should execute. This analysis can actually be strengthed to place more precise bounds on the "jitter" occuring in the execution time of each task – a potentially important property of tasks which communicate with external systems.

- Input-to-output (or "thread") timing requirements; given a sequence of tasks whose execution is necessary for a change in input to propagate to a specific output, we may place overall timing constraints on the time between the execution of the first and last tasks. We can also obtain numerical values from this analysis, giving values for the margins by which each satisfied requirement is actually met.

Future developments in this technique should increase the range of properties which can be specified and quantitative values which can be obtained. Of particular importance is the extension of this work to more complex scheduling strategies. It should be noted that our specifications and analysis do not depend on the form of the scheduler, merely on the fact that a CSP model can be given for it. This fact should certainly facilitiate extending the scope of this work.

B FDR Input Files

The following sections give listings of the input files required to perform this analysis using **FDR**. For clarity, CSP definitions are set in a `teletype` font and comments (lines preceded in the actual files by --) are set in *italic*.

To simplify the maintenance of a range of similar specification processes, some of the specification files have been written to be pre-processed using the Unix m4(1) utility; m4 source files include macro definitions (using the `define` keyword) and instances of these macros, written

```
macro(argument, argument, ...)
```

B.1 The Basic Model (`basics.csp`)

```
include "rtlib.csp"

pragma inline "ML" val _ = print "basics.csp: ";

Nat = {100 * k + 10 *  i + j |
```

```
        i <- {0,1,2,3,4,5,6,7,8,9}, j <- {0,1,2,3,4,5,6,7,8,9},
        k <- {0,1}}

FrameCount = {0,1,2,3}

NumberOfFrames = 4


include "tasks.csp"


pragma inline "ML" val _ = print "Frame interrupt ";
```

Every time signal will represent a fixed interval (determined by the micros scaling function). We must ensure that a frames start at 6250us intervals.

```
pragma channel tock
```

The following process is a simple counter which allows n events on channel a to each one on b.

```
Counter(a,n,b) = Counting(a,n,b,0)

Counting(a,n,b,m) = if   m == 0
                    then b -> Counting(a,n,b,n)
                    else a -> Counting(a,n,b,m-1)

pragma channel framestart
```

`FrameLength = 6250` in us, for a 160Hz frame rate

```
Clock = Counter(tock, micros(FrameLength), framestart)
```

We now define our model of the scheduler. Only one variable influences the tasks scheduled in a frame, its position in the cycle.

```
pragma channel framepos : FrameCount


FrameCounter = framestart -> FrameCounting(0)
```

```
FrameCounting(n) = framepos ! n -> FrameCounting(n)
                   []
                   framestart -> if   n == NumberOfFrames - 1
                                 then FrameCounting(0)
                                 else FrameCounting(n+1)
```

At any time after the start of the first frame this process is willing to output the current cycle count value (n) on the *framepos* channel.

```
pragma inline "ML" val _ = print "Tasks ";
```

The next part of our model is the actual tasks themselves. The execution of a task a will be represented by an event *execed.a* in our model.

```
pragma channel exec : TASKID
```

Whenever we examine a typical task, we read the current scheduler state and choose one of two outcomes: either the task is not to be executed, and we simply move to the next, or the task is due to be run, and so we execute it. In the following definition, i is the task name, d is its maximum duration and C is the set of cycles in which i is executed.

```
TASK(i, d, C) = framepos ? c ->
           if   member (c,C)
           then exec ! i -> RUNNING(d)
           else SKIP
```

A task which is being executed is represented as being able to terminate early (as d above is a maximum duration), or to continue execution until its time left to execute is complete (the variable n represents the maximum time left for the task to run).

```
RUNNING(n) = if   n == 0
             then SKIP
              else (SKIP |~| tock -> RUNNING(n-1))
```

The scheduler loop takes the form of a loop which examines each of the tasks in turn on each cycle. The special task *Background_* is placed at the start of the list of tasks to model the other activity on the processor.

```
Tasks = while(rseq(x, TASKS, CallTask(x)))
```

We include a counter in this task to measure the amount of time available between scheduler cycles. These times will be associated with the current frame position, and output on the channel *slack* (in 100us units)

```
SlackMeasure = {n | n <- Nat, n <= 100}

pragma channel slack :  FrameCount . SlackMeasure

BackgroundTask = Backgrounded(micros(FrameLength))

Backgrounded(cnt) =
        (tock -> if 0 < cnt then Backgrounded(cnt-1) else
                                 Backgrounded(0))
        []
        (framestart -> framepos ? c ->
                slack ! c ! ((micros(FrameLength) - cnt) / micros(250)) ->
                        SKIP)
```

All other tasks are simply treated as non-deterministic periods of computation, as described above. The following definition simply makes extracting the appropriate fields from the data table a little clearer.

```
TASK_INSTANCE(id, data) = TASK(id, fst(data), snd(data))
```

Examining a task now simply involves determining whether it is one of the two special cases:

```
CallTask(id) =
        if id == Background_
        then BackgroundTask
        else TASK_INSTANCE(id, lookup(id, TASKINFO))
```

The overall system model consists of the clock, the tasks and the frame counter composed in parallel. The framepos channel represents local data and is cocealed.

```
System = ((Clock [| {tock,framestart} |] Tasks)
          [| {| framestart, framepos |} |]
          FrameCounter) \ {| framepos |}
```

The actual start of a frame is not required for most specifications, so we may conceal that also:

```
TestView = (System \ { framestart })
```

This definition forms the basis of a variety of checks.

B.2 Real-time Analysis Functions (rtlib.csp)

```
include "tuplelib.csp"
include "arraylib.csp"

pragma opaque "ML" micros
pragma opaque "ML" raiseerror

pragma inline "ML" fun CSPscale [e] =
pragma inline "ML"      Atom (InjectNum ((100 + NumberOf (CheckAtom (e)))
pragma inline "ML"                          div 250));
pragma inline "ML" exception ModelInternalError;
pragma inline "ML" fun CSPraiseerror _ = raise ModelInternalError;

pragma inline "ML" DefineMLFunction "micros" CSPscale;
pragma inline "ML" DefineMLFunction "raiseerror" CSPraiseerror;

pragma inline "ML" ELIDEPRINT.print_elision := SOME 2;
```

B.2.1 Declaration-only version (rtlib.def)

The following declarations can be included in any subsequent file, provided that rtlib.csp has been loaded once.

```
pragma opaque "ML" micros
pragma opaque "ML" raiseerror
```

B.3 Task Data (tasks.csp)

```
pragma inline "ML" val _ = print "task data, ";
```

This file contains the actual scheduler data, stored as an array in execution sequence within the cycle. For each task, it includes a maximum execution time and the set of cycles in which the task is executed.

```
TASKINFO = <
pr(RSD,        pr(micros(500),        {0,1,2,3})),
```

```
pr(RFP,        pr(micros(300),        {0,1,2,3})),
pr(ROT,        pr(micros(250),        {0})),
pr(RAA,        pr(micros(300),        {0,2})),
pr(RWT,        pr(micros(250),        {2})),
pr(RXA,        pr(micros(400),        {1})),
pr(DTM,        pr(micros(250),        {1})),
pr(COT,        pr(micros(250),        {0})),
pr(CSD,        pr(micros(1000),{0,2})),
pr(CIT,        pr(micros(700),        {0,1,2,3})),
pr(CWT,        pr(micros(250),        {2})),
pr(AMX,        pr(micros(400),        {1})),
pr(DI,        pr(micros(500),        {0,1,2,3})),
pr(CFP,        pr(micros(300),        {0,1,2,3})),
pr(AGT,        pr(micros(800),        {1,3})),
pr(DCP,        pr(micros(300),        {2})),
pr(DFP,        pr(micros(300),        {0,1,2,3})),
pr(LSS,        pr(micros(400),        {3})),
pr(IES,        pr(micros(800),        {3}))>
```

The sequence of tasks to be executed starts with the dummy *Background_* task.

```
TASKS = <Background_>^domain(TASKINFO)

TASKID = set(TASKS)
```

B.4 Iteration Specifications (iterations.m4)

```
pragma inline "ML" val _ = print "Loading simple tests: ";

'include' "rtlib.def"
```

Strict version – first event at zero +- limits (thus use limits to allow for phase)

```
BLURRED(i, T_i, L_i, U_i)  = BLUR(i, T_i, L_i, U_i, T_i)


BLUR(i, T_i, L_i, U_i, n) =
(if n < T_i + U_i then tock ->
        BLUR(i, T_i, L_i, U_i, n+1) else STOP)
[]
```

```
(if T_i + L_i <= n then exec.i ->
        BLUR(i, T_i, L_i, U_i, n-T_i) else STOP)



define(iterate,
SpecIter$1 =
BLURRED($1‘,’ micros($2)‘,’ micros($3+($5 * 6250))‘,’ micros($4+($5 * 6250)))
TestIter$1 = (System \ { exec.v | v <- diff(TASKID‘,’{$1}) })
                \ {| framestart‘,’ slack|}

pragma inline "ML" val _ = print "$1 ";
‘divert’(7)
‘iterate_check_command’($1,"SpecIter$1","TestIter$1")
‘divert’(1)
)

define(iterate_check_command,
pragma inline "ML" val Result$1 = CheckTrace $2 $3;
)


f = 6250 permitted fuzz
fl = 0

define(skipiterate)

iterate(RAA,        12500,        0-fl,        f,        0)
iterate(RSD,        6250,        0-fl,        f,        0)
iterate(RFP,        6250,        0-fl,        f,        0)
iterate(ROT,        25000,        0-fl,        f,        0)
iterate(CSD,        12500,        0-fl,        f,        0)
iterate(CIT,        6250,        0-fl,        f,        0)
iterate(CFP,        6250,        0-fl,        f,        0)
iterate(COT,        25000,        0-fl,        f,        0)
iterate(DI,        6250,        0-fl,        f,        0)
iterate(AGT,        12500,        0-fl,        f,        1)
iterate(DFP,        6250,        0-fl,        f,        0)
iterate(RXA,        25000,        0-fl,        f,        1)
iterate(AMX,        25000,        0-fl,        f,        1)
```

```
iterate(RWT,        25000,      0-fl,       f,      2)
iterate(CWT,        25000,      0-fl,       f,      2)
iterate(DCP,        25000,      0-fl,       f,      2)
iterate(LSS,        25000,      0-fl,       f,      3)
iterate(IES,        25000,      0-fl,       f,      3)
iterate(DTM,        25000,      0-fl,       f,      1)

pragma inline "ML" val _ = print "Done\n";
```

B.5 Input-Output Specifications (thrspec.m4)

```
pragma inline "ML" val _ = print "thread specifications, ";
```

This file contains the definitions of processes and macros used in the sequence timing specifications.

```
'include' "rtlib.def"
```

We first define the data values and types used in the measurement of timing margins. *ThreadMaxLimit* is any value at least as great as the longest time limit in a thread specification.

```
MaxLimit = micros(50000)
```

MarginQuantum determines the rounding used when reporting the results.

```
MarginRes = micros(250)
```

The *margin* channel is used to report the results.

```
MarginType = {n * MarginRes | n <- Nat,
                                n <= MaxLimit / MarginRes}

pragma channel margin : MarginType
```

We also require a specification which will allow any trace, and guarantee that all possible executions of the implementation model are explored, in order to ensure that we find all possible values of the timing margin:

```
DF(A) = |~| a:A @ a -> DF(A)
```

```
MarginSpec = DF({|margin|})
```

Now we define some processes used in constructing our specifications. Each specification will consist of two parts, one which keeps track of the elapsed time and another to monitor the calls to specified procedures in order. *reset* is the event used by the latter to inform the timer that the thread has been successfully executed.

```
pragma channel reset
```

The events *irrelevant* and *therest* are used in the following definitions as short-hand for procedure calls which do not satisfy the thread.

```
pragma channel synch

pragma channel irrelevant, therest
```

Limit is the timer half of the specification – when proecdure *trig* is called, it enters the *Bound* state and states counting down the number of *tock* events allowed by the *limit*.

```
Limit(trig,count) = tock -> Limit(trig,count)
                    []
                    exec.trig -> Bound(trig,count,count)
                    []
                    therest -> Limit(trig,count)
```

Bound decerements the timer value whenever *tock* is observed, and simply ignores procedure calls. When a reset occurs, it makes a further synchronization on channel *margin* (this is used in later specifications to allow timing margins to be examined), and then resets to its initial state. If the time limit is exceeded, further *tock* events are not permitted. (This will be visible in practice as a failure of the refinement check.)

```
Bound(trig,count,curr) =
        (if 0 < curr then tock -> Bound(trig,count,curr-1) else STOP)
        []
        reset -> (synch -> Limit(trig,count)
                  [] margin ! MarginRes * (curr/MarginRes) ->
                        synch -> Limit(trig,count))
        [] therest -> Bound(trig,count,curr)
        [] exec ! trig -> Bound(trig,count,curr)
```

The basic unit used to build the part of the specification which monitors procedure executions is $Await(f, X)$. This will perform any event from the set X, and either terminate successfully if the event is f, or remain in the same state if not.

```
Await(f,X) = ([] x : X  @ x -> if x == f then SKIP else Await(f,X))
```

The following macro defines a process which actually implements the specification. It takes 4 arguments: the name of the process to be defined, a trigger event (the name of the procedure which starts the thread), a time limit, and a sequence of procedures which must be called to complete the thread. It instantiates a copy of the *Limit* process together with a sequence of *Await* processes which check for each required procedure in turn. This sequence terminates (and loops back to its initial state) only when all the procedures have been called in order.

```
define(MakeWITHIN,
$1 = ( Limit($2,$3) [[ therest <- exec.head($4) ]]
        [| union({| exec.i | i<-{$2‘,’head($4)}|}‘,’
                 {| reset‘,’synch |}) |]
           while(rseq(y,<exec.$2>^<exec.i | i<-$4 >^<reset>,
                 if y == reset then
                        [] x:{exec.i‘,’irrelevant
                             | i<-union({$2}‘,’set($4))} @
                                x -> reset -> synch -> SKIP
                 else
                 Await(y, union({irrelevant}‘,’
                                { exec.i | i<-union({$2},set($4)) }))))
         ) \ {reset}
)
```

The *reset* event is not intended to be part of the specification, and thus is hidden from the environment.

Finally, we define a macro which builds up the refinement test for a given thread. The parameters are a thread name or number, a starting procedure, a time limit and a list of other procedures. It defines a copy of the specification (and a version for compliance testing which has the margin channel hidden) and a test process which takes our system model and conceals all events which are not relevant to the particular thread.

The final group of lines may be used to add extra lines to the end of the pre-processed output to run each of the tests in turn.

```
define(thread,
MakeWITHIN(Single$1,$2, micros($3),$4)

Spec$1 = Single$1 \ {synch}

Test$1 = (System [[ exec.i <- irrelevant
                  | i <- diff(TASKID‘,’ union({$2}‘,’set($4))) ]])
         \ {| framestart‘,’ slack |}

pragma inline "ML" val _ = print "$1 ";

‘divert’(7)
‘check_command’($1,"Spec$1","Test$1")
‘divert’(1)
)

define(check_command,
pragma inline "ML" val Result$1 = CheckTrace $2 $3;
)
```

B.6 Input-Output Specification Data (thrdata.m4)

```
‘include’ "rtlib.def"

include(thrspec.m4)
include(margins.m4)

define(skipthread)
```

The fuel pump is driven correctly within 6.25ms of a detected change in pressure:

```
thread(a, RFP, 6250, ‘<CFP, DFP>’)
```

The injector timing is adjusted within 6.25ms of a change in reported accelerator angle:

```
thread(b, RAA, 6250, ‘<CSD, CIT, DI>’)
```

Exhaust analysis changes propagate to injector timing within 25ms:

```
thread(c, RXA, 25000, '<AMX, LSS, CSD, CIT, DI>')
```

Changes in reported engine speed are reflected in injector timing in 6.25ms

```
thread(d, RSD, 6250, '<CIT, DI>')
```

Cooling-water over heating is indicated to the driver in 12.5ms(!)

```
thread(e, RWT, 12500, '<CWT, IES>')
```

Fuel pressure changes may take nearly a whole cycle to reach the driver: (who, fortunately, won't be able to respond in 25ms anyway!)

```
thread(f, RFP, 25000, '<CFP, IES>')
```

Coolant temperature changes are reflected by the pump (fan?) in 6.25ms

```
thread(g, RWT, 6250, '<CWT, DCP>')
```

Excessive oil temperature will be transmitted to the speed-demand controller within 6.25ms

```
thread(h, ROT, 6250, '<COT, CSD>')
```

And reported to the driver in 25ms

```
thread(i, ROT, 25000, '<COT, IES>')
```

The tachometer is never more that 25ms out-of-date

```
thread(j, RSD, 25000, '<DTM>')
```

Changes in demanded speed propagate to ignition timing in no more than 12.5ms

```
thread(k, RAA, 12500, '<CSD, CIT, AGT>')
```

Changes in actual speed propagate to ignition timing in no more than 12.5ms

```
thread(l, RSD, 12500, '<CIT, AGT>')
```

Limits in speed due to excessive oil temperature similarly take no more than 12.5ms to influence ignition timing

```
thread(m, ROT, 12500, '<COT, CSD, CIT, AGT>')
```

Exhaust analysis results propagate to ignition timing (a perhaps somewhat convoluted path in ...? A single cycle ?

```
thread(n, RXA, 25000, '<AMX, LSS, CSD, CIT, AGT>')
```

Or somewhat longer?

```
thread(ndash, RXA, 31250, '<AMX, LSS, CSD, CIT, AGT>')
```

```
pragma inline "ML" val _ = print "Done\n";
```